07/10/2018

Liceo delle Scienze Applicate

Computer Science

Fifth course

Computer graphics, and beyond

Topics:

- Introduction to Tkinter
- Geometry
- Curve tracing
- GIS
- Systems and simulations
- Cellular automata
- Turing machines
- Combinatorics
- Fractals
- Appendices

"… WITH GREAT POWER THERE MUST ALSO COME -- GREAT RESPONSIBILITY."

(Stan Lee)

Thanks to:

Adriano Barlotti (1923-2008), geometer extraordinaire 

Guyon Roche, Web developer 

Mike Geig, IT instructor at Stark State College (OH), gamer 

"Debugging is twice as hard as writing the code in the first place. Therefore, if you write the code as cleverly as possible, you are, by definition, not smart enough to debug it."

Brian W. Kernighan (1942-)

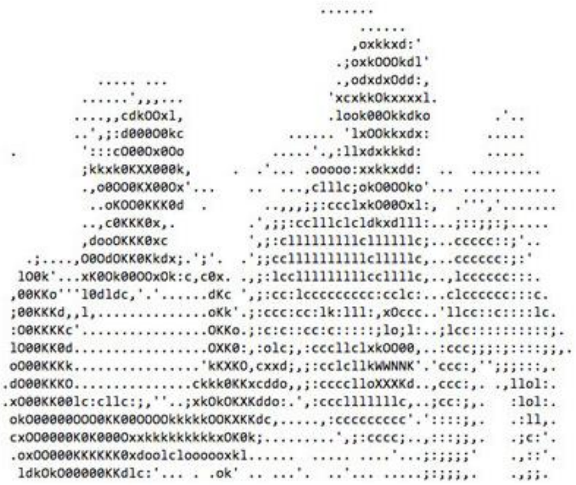
Tkinter

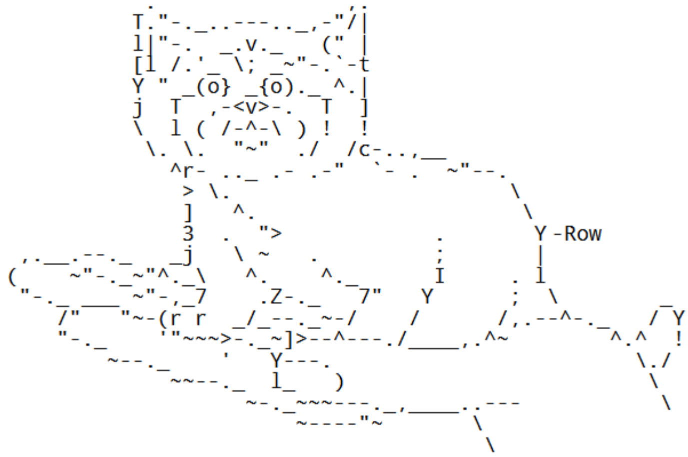
a graphics library for Python

Topics Geometry

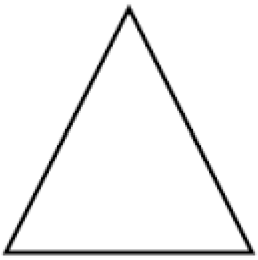
Drawing with computers

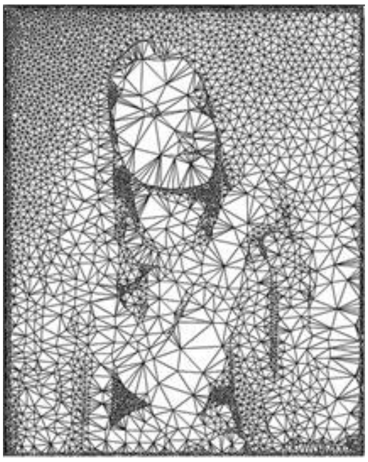
Computer graphics has come a long way since images were approximated with ingenuous combinations of characters:

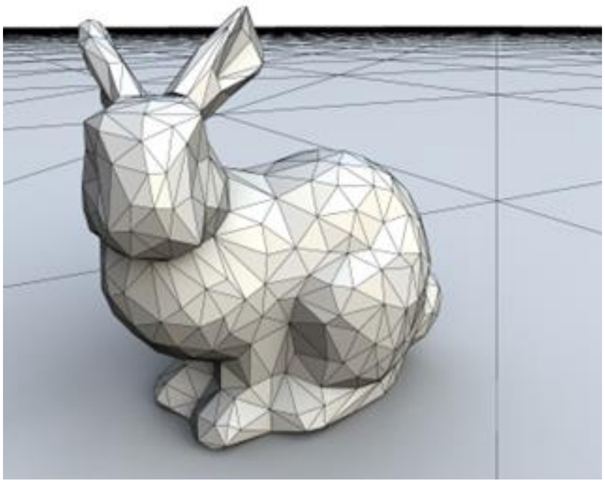




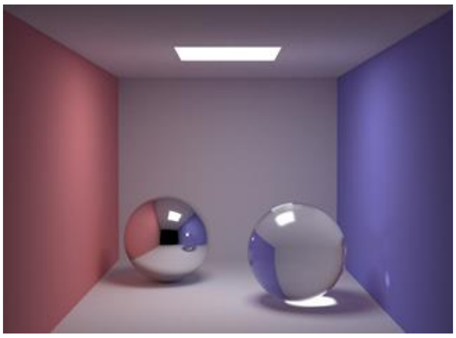
It all starts with a bit of geometry: in fact, the basic unit for computer-generated images is the *triangle*, which works in 2D and 3D graphics:







With a good resolution, the results can be quite remarkable:



When geometry is set in motion, it becomes *animation* (more on this later).

In this course, the programming language of choice is **Python**, which comes bundled with a well-known graphic library called **Tkinter**, which is present in both versions 2 and 3:

Python 2 Tkinter logo

Python 3 Tkinter logo

Installing *Python*

Of course, those who have already installed Python can skip this part.

The Python compiler is available at several Web sites, the official download for Italy page being:

`http://www.python.it/download`

Guido van Rossum, inventor of Python

Quite naturally, one should choose the installer corresponding to the computer in use. As of July 2018, the latest versions available were the 3.7.0 and the 2.7.15. It may appear strange that version 2 is still being developed along with version 3 (its latest release dating to May 2018), but there are still some computers which do not support Python 3.

On Windows systems, it is suggested to choose a "customized" installation, but only the first of the following suggestions is strongly encouraged (the others are needed only if one plans to install external libraries, which are not treated in this course):

- Add Python to **`PATH`** (environmental variables: first window)
- Install **`PIP`** (should be automatic)
- If the user plans to install external libraries, it is best to install Python into a hard disk folder *with a simple name* (e.g.: **`c:\Python37`** or anything similar) instead of the suggested location in "**`Programs`**": this will make further installations easier. Otherwise, the default installation folder can be used.

Who is the man depicted here?

Programming languages are usually not equipped for computer graphics, forcing developers to use external libraries, a somewhat non-standard practice that can bring both installation and compatibility problems. **`Tkinter`**, written by Fredrik Lundh (a Swede), is an exception, being bundled in the **`Python`** environments.

As with other features (notably, console **`I/O`** functions), there are inexplicable differences between versions **`2`** and **`3`**. For this reason, the **`Tkinter`** library must be imported with slightly different names, as in:

```
from sys import *
if sys.version_info.major == 2:
    from Tkinter import *
elif sys.version_info.major == 3:
    from tkinter import *
```

This snippet works in both versions of **`Python`**; it is still possible to use the directive suited to one's version.

This course requires two distribution files, which provide tools for geometric programs:

`marioGraphics.py / marioConsole.py`

These libraries (which will be introduced shortly) *must* be stored in the same folder as the **`Python`** source files[1]. All the import directives (both for **`Tkinter`** and **`marioConsole`**) are in **`marioGraphics`**, so the following directive suffices for covering all the possibilities in any **`Python`** program:

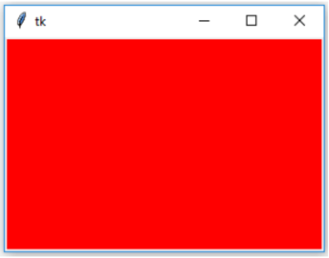
```
from marioGraphics import *
```

[1] This is the simplest way of ensuring that said files can be *imported* into new programs. A more complicated possibility is to manipulate the Python setup, but it is strongly deprecated, if only for the sake of simplicity.

First steps

The first example is presented "as is", but it should not be difficult to follow. It sums up the most basic settings necessary to display what is called a *graphic window* (also known as *viewport* or *canvas*).

```
from ... import *       #  importing the version of choice (tkinter/Tkinter)
import time             #  it is also possible to use only marioGraphics

window = Tk()
screen = Canvas(window,bg="#FF0000",width=300,height=200)
screen.pack(expand=True,fill=BOTH)
screen.update()
time.sleep(4.6)
window.destroy()
del window
```



The result is a blank display window (**300** by **200** pixels wide, with a **red** background), which automatically disappears after **4.6** seconds (a so-called *timeout period*).

Let us now examine the directives and the statements of this program.

Libraries	The **Tkinter** library contains the basic graphic functions. Complex programs will need many more settings (more on this soon). **time** is needed for the timeout operation at the end of the program.
The root window	A program using **Tkinter** must create a multi-purpose window, called a **Tk()** object, which can contain one or more secondary windows called ***frames***. The main window is assigned to a specific variable, normally called a *handle*, that is, a reference item for further operations on the window (which are very infrequent indeed, possibly limited to its destruction and deletion at the end of the program).
The graphic window	The graphic window on which **Tkinter** draws is managed separately from the console window (this is quite common with graphic libraries). It can be declared inside the main **Tkinter** window, and it, too, is associated to a specific variable, a **Canvas** object, which, on declaration, can be assigned its characteristics. The variable can be used as a *handle* for further operations on the viewport, as can be seen when it is *packed* to occupy all the space in the root window.
Configuring the canvas	The **bg** (background) property of a **Canvas** object can be defined (in 16M color depth) with the respective values for red, green, and blue, in 2-digit hex format (**00** through **ff**, or **0..255**), with a leading pound (**#**) sign, as in: **#rrggbb**. The assignment of dimensions (width and height in pixels) is quite straightforward.
Showing data	Graphic programs use two bitmaps: one is for processing (storage buffer), one for display (the actual map of the graphic window). Among other things, this makes it possible to prepare a complete stage before showing anything, because every display operation works on the storage buffer, not on the slower display. After the call to **update()**, **Tkinter** swaps the two bitmaps, showing the actual content at RAM speed (taking just a few microseconds). It must be noted that **Tkinter** is not exceedingly fast: it has been chosen for its simplicity of installation and use, not for its performance, while other libraries, though faster, are not so easy to set up. Then again, **Python** itself, being interpreted, is not fast in the first place.
Freezing the program	The **time.sleep()** function stops program execution for a given interval, expressed in seconds, as a decimal number: **4.6** means four and 6/10 seconds. In later chapters, other delay functions will be considered.
Destroying the canvas	In most **OOP** environments, objects are normally destroyed automatically as soon as the program reaches its end, but in the case of **Tkinter** this does not happen, so the **destroy** call is necessary for the graphic window to disappear. Anyway, nothing serious is going to happen if one forgot this step: further executions of programs using **Tkinter** will set up a new viewport in place of the old one.

Tkinter and geometry

Animation (a subject for further chapters) is the core of gaming programs, and is normally used to simulate[2] real or imagined situations on the canvas. In its simplest form, it is just *geometry in motion*: so, our first step will be to familiarize with geometry in **`Python`** programs, through the use of **`Tkinter`**.

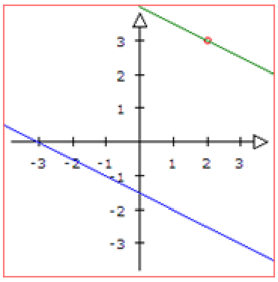
In computer graphics, *geometric primitives* (or *graphic primitives*) are functions that display geometric elements (also called *shapes*, or *geometric objects*) on the screen. The representation, for now, will only be static, that is, without moving elements around (more on animation later).

All graphic libraries support the simplest shapes:

- Points[3]
- Lines and segments
- Circles and ellipses[4] (known as *ovals*)

In the case of **`Tkinter`** (and other libraries), slightly more complex shapes are also supported:

- Rectangles
- Polygons[5]
- Polylines (sets of contiguous *segments*)



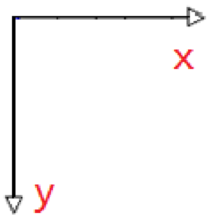
There are other miscellaneous elements, some strictly geometric, some not:

- Arcs
- Raster images (basically, these are large Point objects with many pixels)
- Text (these are large Point objects, too)

Even in simple geometric problems, many of the mentioned elements are present, as we can see in the image to the right, representing a classic problem in analytic geometry (searching a **line** parallel to **another**, passing through a given **point**).

The *point* **`(2,3)`** is rendered as a small *circle*, or it would be hardly visible as a single pixel[6]; the parallel lines, the coordinate axes, and even the tick marks on the axes are all *segments*; the arrows on the axes are *polygons*; the captions near the axes are *text*.

`Tkinter`, like most graphic libraries, also offers the possibility of specifying a color and a thickness for any shape drawn (for text, it will be the dimension), and the choice of filling closed shapes (ellipses, rectangles, and polygons) with a solid background. These actions only require additional clauses or statements.



The main geometric issue is correctly placing the geometric elements (except, of course, audio clips) on the canvas, which raises two problems: the first is that pixels are identified by a coordinate system with the y-axis pointing down and the origin **`(0,0)`** lying in the top-left corner; the second is that pixels, by their nature, are whole entities, while geometry requires specifying coordinates as real (in computer terms, *double*) numbers. The screen, besides being reversed vertically, has neither negative nor fractional coordinates, unlike our habitual Cartesian plane.

So, we end up with two coordinate systems, one for the object of our study, the other for the graphic window, called, respectively, *world* and *screen* coordinates. The process of matching world and screen is called *calibration*, which is not as difficult as it may seem: after all, it only takes a transformation of coordinates and two scaling proportions (understandably, with a bit of rounding). More on this later.

`Tkinter` does not support audio, which is not an issue in this course (although it would add some spice to animations). That would require other, more complex, libraries.

---

[2] There are several meanings of the word *simulation*. In this course, the term will be restricted to “moving images on a computer screen”.

[3] Points should be represented with single pixels, but, being quite difficult to see, a single geometric point will be enhanced by drawing a small disk, just to make it visible. Large numbers of points, though, will preferably be represented with single pixels, to avoid cluttering the viewport.

[4] It is true that circles *are* ellipses: yet, these elements are often treated as different entities, so that the resulting geometric redundancy actually simplifies the work of the programmer.

[5] Triangles and rectangles (and, less obviously, segments) are *polygons*: the same considerations done on circles and ellipses apply.

[6] As already hinted, when large numbers of points are needed, they can be displayed as single pixels.

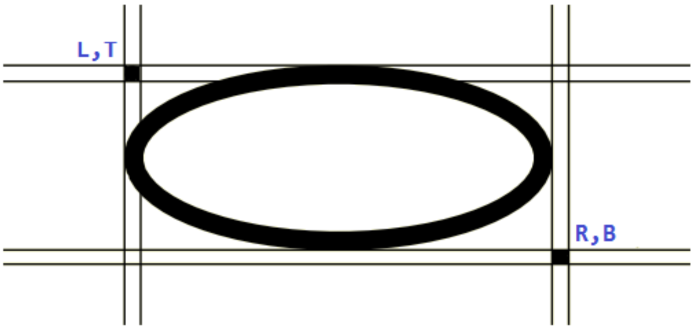
Colors

Tkinter colors are mapped through their **red**, **green**, and **blue** components (in hex format), with values ranging from **00** through **ff** (**255**), and preceded by a pound sign. For example, the color **DeepPink** corresponds to the string **#ff1493**.

Surfaces

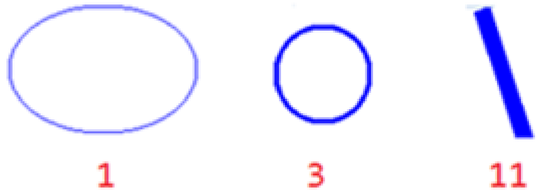
Ovals and rectangles are defined by specifying the rectangular area that they occupy on the canvas:

```
screen.create_oval(L,T,R,B)        #   The R,B pixels are not included
```



Thickness

For linear elements, it is possible to specify the brush thickness in pixels, which is an integer value (after all, as already mentioned, pixels are integer entities). A hairline brush has thickness either **1** or **0**; negative thickness values generate an error, e.g.: **bad screen distance "-6"**.



Filled shapes

Ovals, rectangles, and polygons are filled shapes, and they can be assigned a **fill** color:

```
screen.create_oval(30,170,60,100,fill='orange',outline='red')
```

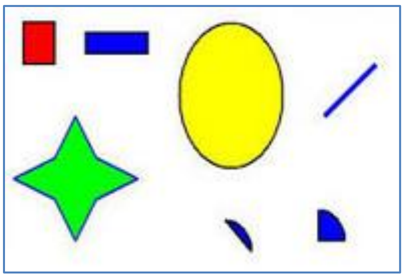
General format of graphic primitives

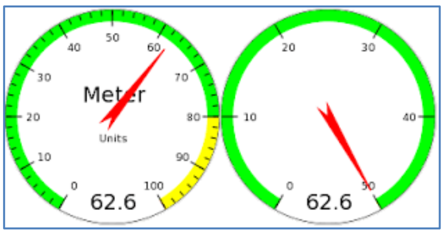
Tkinter graphic primitives follow a common format (of course, there are distinctions, as different elements require different specifications). They all applied to a **Canvas** object, they start with **create_**, followed by a mnemonic:

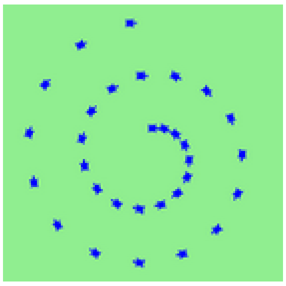
```
screen.create_x(<coordinates>,<options>)
```

x could be the name of any geometric shape[7]; the options normally include width and colors.

An incomplete list of shapes (excluding some which require further specifications, and some others which will not be treated in this course) will be shown in the next section.





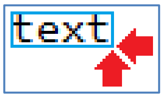


[7] An incomplete list: **line**, **oval**, **rectangle**, **polygon**, **arc**. For single pixels, the solution is using a **1**-pixel square (which is, after all, just a rectangle with both sides set to **1**).

Examples

In general, shapes are defined with the coordinates of the rectangle they occupy on the screen. Images, of course, depend on their size, and can be scaled up or down (though not as efficiently as with other graphic libraries); text elements, understandably, depend on font and size: both elements are, by default, centered (or *anchored*) around a reference point, in **9** different modes (**CENTER** being the default):

NW N NE W CENTER E SW S SE

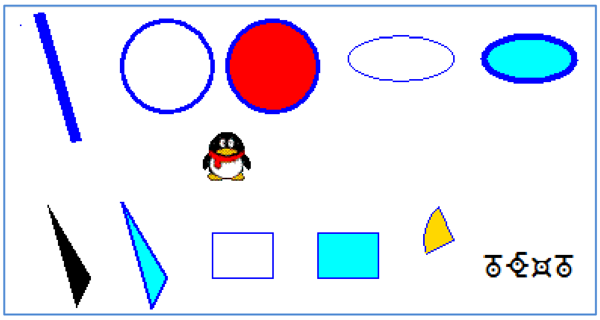


For example, **SE** will position the element so that the coordinates of the *reference point* will mark the right-bottom (south-east) corner of the box containing the text (see image).

The following example (where the canvas has been named **s** for brevity's sake) displays some elements, also showing how everything is kept on hold until a call to **update()** dumps the storage buffer onto the screen, showing the actual content in one fell swoop[8]. The playing field can thus be displayed only after all the graphic entities have been put into place (an essential technique in animations, where it is fundamental to give the sensation of simultaneous movement).

```
s.create_line(13,13,14,14,fill="blue")
s.create_line(25,5,50,90,fill="blue",width=8)
s.create_oval(80,10,140,70,outline="blue",width=3)
s.create_oval(150,10,210,70,outline="blue",fill="red",width=3)
s.create_oval(230,20,300,50,outline="blue")
s.create_oval(320,20,380,50,outline="blue",fill="aqua",width=4)
s.create_polygon(30,130,60,180,50,200
s.create_polygon(80,130,110,180,100,200,outline="blue"",fill="aqua",width=2)
s.create_rectangle(140,150,180,180,outline="blue")
s.create_rectangle(210,150,250,180,outline="blue",fill="aqua")
s.create_arc(280,130,320,180,outline="blue",fill="gold",start=120,extent=80)
s.create_text(350,170,font=("Annone",20),text='text')
img = PhotoImage(file="mario1.gif")
s.create_image(150,100,image=img)
s.update()
```

The result is the following, where the real challenge is seeing the pixel (to the immediate left of the thick segment, in the top-left corner):



WARNING: the font used in this example is Annone (or [illegible]); any font must be regularly installed on the system, or the text will default to a rather stale **Arial**.

The following apply:

- A single pixel is drawn by using a **1x1** segment;
- Ellipse, rectangles, and arcs must defined in a *rectangular* portion of the viewport;
- Polygons can have any number of arguments (provided they represent a sequence of coupled coordinates): for example, a triangle is a polygon defined by six numbers; with two points, the polygon collapses to a segment;
- Text elements can use any legal font and size (but the font must be registered in the system);
- Images must be loaded into memory with **Photoimage** before being created;
- **Tkinter** for **Python 2** only supports **GIF** images;
- **Tkinter** for **Python 3** only supports **GIF** and **PNG** images;
- Animated images are not supported.

[8] *One fell swoop*: a single, sudden action. The first documented reference of this expression is found in Shakespeare's *Macbeth* (1605), as spoken by MacDuff. The word *fell* used to be associated with cruelty (as in *felon*), but it has now lost that connotation, being only used in this idiom.

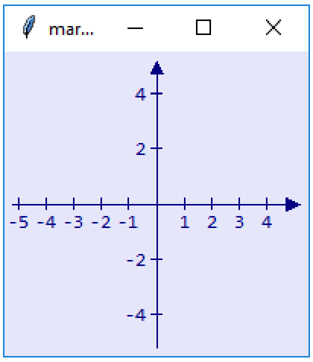
Geometry with marioGraphics

Tkinter GIS

A designer object

Provided the programmer knows the Cartesian plane, **Tkinter** is not too difficult to master, and works similarly to all graphic libraries, supporting the most common **2D** shapes; however, it is a long way from a working tool for geometry, being only a *viewer* with a restricted coordinate system. What we need is a set of functions for the most common geometric tasks, ranging from calibration to the management of entities like points, lines, circles.

For this reason, this course will refer to a *geometrical* library, based on **Tkinter**, where the majority of basic plane geometry problems are already solved. For example, it calculates which line lies on two given points, or the dual problem of which point lies on two given lines; further functions find the intersections of curves with one another; with calibration, any world coordinate system can be mapped to any graphic window, also drawing the coordinate axes, if need be. It also can add animation: it makes it possible to write a function tracking the movement of objects in a real or simulated world, obeying any given law of physics, then reproducing the situation on the screen with a program loop (a process called *simulation*). The library (**marioGraphics.py**), along with a supplemental file[9] (**marioConsole.py**), should be placed in the same folder as the source files, so as to be included into it with an **import** statements.



The functions of this library dispose of any considerations relative to the preparation of a graphic window such as: defining the window dimensions, calibrating the viewport, setting foreground and background colors, and sketching the coordinate axes (with demarcation notches). All these operations are activated with calls to the designer, without having to know all the internal details of **Tkinter**. The resulting viewport displays a portion of the Cartesian plane, as shown in the image to the right.

Besides **Tkinter**, these activities require scores of functions, but the main program is quite slim, the details being hidden in the libraries. These functions are methods of a *designer object*, packed with several graphic routines.

The first complete example (introduced *as is*, without much ado) is the following:

```
from marioGraphics import *                 #   reference to geometric library
d = Designer()                              #   declare designer object
d.dimensions(200,200,-5.5,5.5,5.5,-5.5)     #   define window and calibration
d.foreground("#0000ff")                     #   set foreground color (BLUE in hex)
d.background(LAVENDER)                      #   set background color (color NAME)
d.axes(1,2)                                 #   sketch coordinate axes (w/notches)
d.freeze()                                  #   wait for user action (Esc key)
d.destroy()                                 #   destroy viewport
del d                                       #   destroy designer
```

NOTES

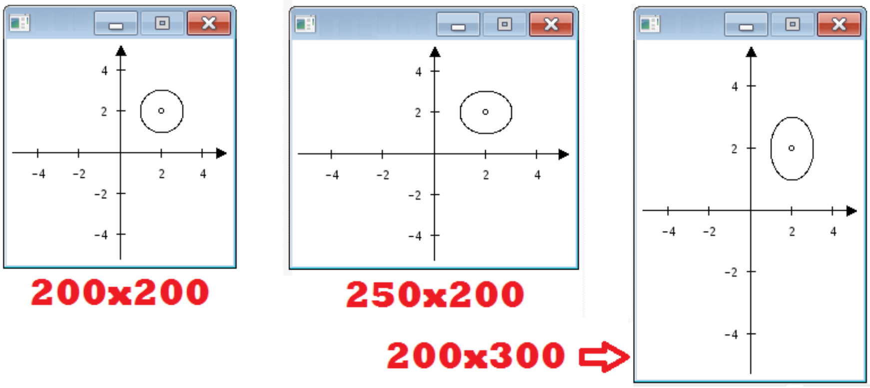
- The arguments of the **dimensions()** method are, in order: the size of the graphic window (width and height), and the limits of the world window (in the classic sequence: **left**, **top**, **right**, and **bottom**);
- The **foreground** and **background** methods can use the **CSS** color names (known from previous courses), in **UPPERCASE** letters, when not between quotes, or with no such restrictions when between quotes (as in **"rosyBrown"**); a complete list of color names is shown in **appendix A**;
- colors can still be defined in **RGB** format (in hex digits, between quotes with a pound prefix);
- The resulting sketch is isometric only because of the chosen calibration, which matches a square world to a square graphic window: different choices would result in stretched views;
- The two arguments or the **axes()** method represent the spacing of the marker notches (first in **x**, then in **y**): in this case, the factor for the **y** axis has been set to **2**, but this is only an example; the spacing heavily depends on the world dimensions.

The sample program uses lines, filled triangles, and text, without the need of directly referring to the corresponding **Tkinter** functions, thus simplifying the task of the programmer. In a similar fashion, geometric entities like points, lines, circles, and the like can be defined and drawn with calls to the designer: **Tkinter** will only work from behind the scenes.

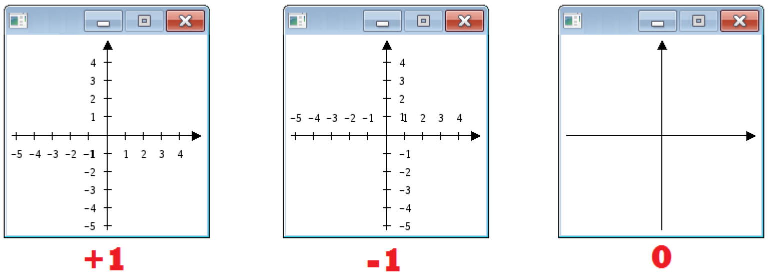
[9] The supplemental file contains the basic display routines (that is, the **print** function), which, for reasons unknown, are incompatible between **Python** versions **2** and **3**.

Variations on the coordinate axes

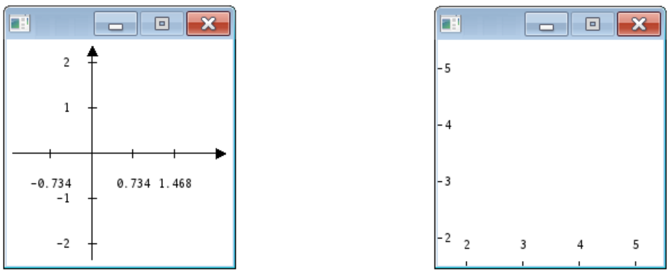
The choices of graphic window and calibration influence the outcome of the view. In the three examples below, the same world has been calibrated against different graphic windows, and a circle, distorted or otherwise, shows the different effects:



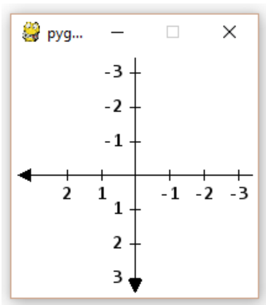
The scaling coordinates along the axes can be positioned on either side, by calling the **`axes()`** method with positive or negative arguments (or with a mix of the two). A zero value, though allowed, would hardly be useful, as no scale would be visible. The default parameter value is **`+1`**. Each axis could be drawn independently, with methods called, respectively, **`xAxis()`** and **`yAxis()`**.



These methods try to compensate for unreadable axes: if the calculated lines were to be too densely packed, the scaling factor could be expanded automatically. In the following example, the **`y`** scale was given as **`0.01`**, but the method amplified it to **`1`**, in order to have readable coordinates. The first example also shows a horizontal displacement of **`0.734`** (quite useless, but just to show that it can be done). The second example shows how the designers renders an axis lying outside the world (that is, on either side of the viewport, depending on its relative position):



It is also possible to set a custom orientation for either axis, just by setting the left of the world greater than the right (the same goes for the top and the bottom). In the example, the calibration used was totally reversed, with **`3.6,-3.6,-3.6,3.6`**, respectively, as left, top, right, bottom:



DO NOT FORGET

When writing programs with the **Tkinter** library, there are some details which must be taken care of. The following is a summary of the necessary steps.

1 The step of activating **Tkinter** can be omitted, since the designer object automatically provides for this when it is initialized.

2 The designer object should (or, almost certainly, *must*) be calibrated, involving the dimension of both *world* and *screen* windows; there are several ways to proceed.

On declaration, the default window is **400x400**; calibration can follow afterwards:

```
d = Designer()
d.calibrate(-1,0.6,2.2,-0.6)
```

Also on declaration, both windows can be specified, and calibration is automatic:

```
d = Designer (200,100,-1,0.6,2.2,-0.6)    #    on declaration
```

One method of the designer can specify all the details:

```
d = Designer()
d.dimensions(200,100,-1,0.6,2.2,-0.6)    #    after declaration
```

The two steps can be taken separately:

```
d = Designer()
d.dimensions(300,300)
d.calibrate(-1,0.6,2.2,-0.6)
```

3 If needed, the axes (one or both) can be drawn, jointly or separately (with or without captions):

```
d.axes(...)                  #   jointly

d.xAxis(...)                 #   separately
d.yAxis(...)

d.axes(0)                    #   no captions
```

4 The default color set is black on white, but the colors can be changed at will; the **Tkinter** specification is by the **RGB** string format, but the designer object allows for the HTML color names (either as strings or as names, in which case they must be in UPPERCASE letters):

```
d.foreground(0,255,255)      #   three integer values 0..255
d.foreground((0,255,255))    #   a tuple with three integer values 0..255
d.background(SNOW)           #   named color, in UPPERCASE letters
d.background("Pink")         #   string color, no set case
```

5 Program execution can be suspended indefinitely (until the **Esc** key is hit or the “close” button of the viewport is clicked), with the following method:

```
d.freeze()
```

6 Other methods mimic the corresponding **Python** function (suspending the program for a given interval, expressed in seconds):

```
d.rest(2.5)
d.wait(2.5)
d.sleep(2.5)                 #   totally equivalent
```

7 Whenever the canvas requires being cleared, this can be done with the following method:

```
d.clear()                    #   uses the current background color
d.clear(GREEN)               #   changes the background color
```

This is especially done during *animations*, where scenes are redrawn continually.

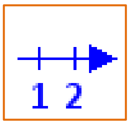
8 When terminating the program, the **Canvas** and **Tkinter** objects associated to the designer should be deleted: forgetting this step would leave the viewport on the screen. This would only be a minor nuisance, however, since running any subsequent **Python** program would automatically remove it.

```
d.destroy()
del d
```

Common problems with calibration

Calibration is never perfect (on a finite machine in this vale of tears, nothing is). Mapping any world to any viewport invariably carries a bit of approximation, as we will see in the following example.

The test was made with a tiny viewport (48 pixels), and only the **x** coordinates are examined, since every consideration can be analogously applied to the **y** axis.



The viewport is **48** pixels wide (**0** through **47**), and the world boundaries are **0.15** and **3.6**. The declaration was:

```
a = Designer(48,48,0.15,3.2,3.6,-3.2)
```

After calibration, the properties of the designer are:

```
g.l    0.15
g.r    3.6
g.px   0.07340425531914894
g.x1   13.6231884057971

g.t    3.2
g.b    -3.2
g.py   0.13617021276595745
g.y1   7.34375
```

g.px is the world distance corresponding to one pixel; **g.x1** is the number of pixels corresponding to one world unit.

The vertical properties are not relevant, but are shown for completeness (the data are different because the calibration is not isometric).

The columns to the right show the **x** world coordinate for each of the **48** pixels of the viewport.

We can see that one pixel corresponds to about **0.073** world units, while one world unit corresponds to about **13.62** pixels. The problem is that pixels are integer quantities, so it only makes sense that screen coordinates and distances may show a little variance from their actual values. For example:

- coordinate **1** is approximated at the **12**th pixel;
- coordinate **2** is approximated at the **25**th pixel;
- coordinate **3** is approximated at the **39**th pixel.

The approximated *world* values for those pixels, though, are, respectively, **1.03085**, **1.985106**, and **3.01277**. Also, **1** and **2** are **13** pixels away, while **2** and **3** are **14** pixels away, rounding to different integers the **13.62** calibration figure.

But that's a fact of life in computer graphics, since floating-point arithmetic has a limited number of significant digits[10]: In most programming languages (for example, in **Python** and **C++**) that number is **15**, although numbers are stored with **17** (the last two digits, though present, are not reliable).

UNIT	PIXEL	WORLD COORDINATE
	0	0.1500000000000000
	1	0.2234042553191489
	2	0.2968085106382979
	3	0.3702127659574468
	4	0.4436170212765957
	5	0.5170212765957447
	6	0.5904255319148936
	7	0.6638297872340426
	8	0.7372340425531916
	9	0.8106382978723404
	10	0.8840425531914894
	11	0.9574468085106383
1	12	1.0308510638297870
	13	1.1042553191489364
	14	1.1776595744680851
	15	1.2510638297872340
	16	1.3244680851063830
	17	1.3978723404255320
	18	1.4712765957446807
	19	1.5446808510638297
	20	1.6180851063829786
	21	1.6914893617021276
	22	1.7648936170212766
	23	1.8382978723404255
	24	1.9117021276595740
2	25	1.9851063829787234
	26	2.0585106382978730
	27	2.1319148936170220
	28	2.2053191489361703
	29	2.2787234042553193
	30	2.3521276595744682
	31	2.4255319148936167
	32	2.4989361702127660
	33	2.5723404255319150
	34	2.6457446808510640
	35	2.7191489361702126
	36	2.7925531914893615
	37	2.8659574468085110
	38	2.9393617021276595
3	39	3.0127659574468084
	40	3.0861702127659574
	41	3.1595744680851070
	42	3.2329787234042553
	43	3.3063829787234043
	44	3.3797872340425530
	45	3.4531914893617020
	46	3.5265957446808510
	47	3.6000000000000000

[10] After all, "We live in the world of approximation: we delete terms from equations" (Math wisdom).

Geometric objects

As previously mentioned, the library contains built-in functions for the management of several geometric objects (or *shapes*, in computer graphics parlance).

The most common objects are the following (as they would be declared in a **Python** program):

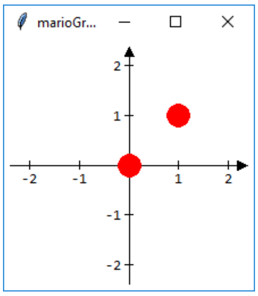
```
p = Point()                    #    the basic geometric object
q = Pixel()                    #    similar to Point, one screen pixel
s = Segment()                  #    defined by two end points (a and b)
c = Circle()                   #    defined by center and radius
r = Line()                     #    considered "infinite" (more or less)
i = Image()                    #    an image is treated like a point
z = Rectangle()                #    defined with: left, top, width, height
```

The difference between **Point** and **Pixel** is that the former is rendered as a small circle, and the latter as a true screen pixel: as mentioned before, classic geometric problems, which feature few points, require them to be easily distinguishable on the screen, while problems in other sciences (for example, physics, or statistics) show large collections of points, which form geometric shapes by the strength of their number.

As its name implies, the **Image** object can be associated to any image file. It is used for sprites, animated or otherwise, and in all geometrical considerations it is treated as a **Point** (that is, defined by two coordinates, around which the image is centered). A couple of other shapes are not treated in this course (**Disk**, **Ray**).

Besides the first geometric object, the following example will introduce some of the features of the library. The first geometric object shown is the **Point**, which, understandably, has two coordinate properties (**x** and **y**, both **numeric**). In the example, points are drawn in **red**, with an exaggerated thickness of 10 pixels, just to show that it can be done: the default thickness is 3 pixels (corresponding to the radii of the circles).

```
d = Designer()                 #    default: 400x400
d.calibrate(-2.5,2.5,2.5,-2.5) #    late calibration
d.axes(1,1)
a = Point()                    #    default: 0,0
d.draw(a,RED,10)               #    designer method (with color/thickness)
a.x+=1                         #    move point
a.y+=1
a.draw(d)                      #    object method
```



The result is shown in the image to the right.

The graphic library makes some assumptions, so as to simplify the program by the use of some common settings; in other cases, the standard drawing conventions can be modified to obtain a better view:

- The default viewport size is **400x400**;
- Colors (both foreground and background) can be set with the **RGB** quantities (in hex), the HTML color names (in **UPPERCASE**; see **Appendix A** for a complete list), or as a string (with no set case: "**Pink**" is as good as "**pink**"); the default combination is black on white;
- The default coordinates of a **Point** object are **(0,0)**, corresponding to the origin;
- Objects can be sketched with the **draw** method: the designer object and all geometric objects have it, so that it can be sent to the designer, passing the object, or to the object, passing the designer; colors are suggested to mark different phases of a problem (e.g: given shapes in **blue**, results in **red**)
- The **draw** has additional parameters for the setting of color and thickness for any shape.

Methods are commutative with regards to the objects and the arguments: it could be **g.draw(a,RED)**, or **a.draw(g,10)**, or **g.draw(a,10,RED)**. These features have been added to simplify the life of the programmer, who is not forced to remember a specific order of the arguments.

WARNING !!! The coordinates can be changed at will, but this does not mean that the object will move: what is on the screen stays on the screen, unless a **clear()** method is issued to the designer, in which case everything has to be redrawn.

In some cases, though, shapes already present in the canvas can be *moved* without affecting the rest of the viewport (more on this in the chapter on *animations*).

Initializing shapes

Object attributes can be initialized at their declaration, in a variety of ways:

Declaration	Resulting coordinates
`p1 = Point()`	`0,0`
`p2 = Point(x,y)`	`x,y`
`p3 = Point(x)`	`x,0`
`p4 = Point(p)`	`same as` *`point`* `p`

A **`Pixel`** is a special type of **`Point`**, shown as a single pixel on the viewport.

Most other objects treated with this library are *vector* objects, since they are identified by their *geometrical*, rather than *graphical*, properties (which are only used on the screen).

For example, a **`Segment`** objects contains two **`Point`** objects as properties, named **`a`** and **`b`**. Every segment has them; they, in turn, contain two **`double`** properties (**`x`** and **`y`**). This means that, when declaring two **`Segment`** objects **`d`** and **`e`**, the program will contain four **`Point`** objects (**`d.a`**, **`d.b`**, **`e.a`**, and **`e.b`**) and eight "double" values (the numbers **`d.a.x`**, **`d.a.y`**, **`d.b.x`**, **`d.b.y`**, **`e.a.x`**, **`e.a.y`**, **`e.b.x`**, and **`e.b.y`**).

`Segments` (and **`Dashes`**) have several possible initializations the assumption here with segments is that unspecified *coordinates* default to **`zero`**, and unspecified *points* default to the **`origin`**:

Declaration	Resulting coordinates
`s1 = Segment()`	`0,0/0,0`
`s2 = Segment(a,b,c,d)`	`a,b/c,d (coordinates of the end points)`
`s3 = Segment(p1,p2)`	`s3.a: same as p1; s3.b: same as p2 (type` *`point`*`)`
`s4 = Segment(s1)`	`s4.a: same as s1.a; s3.b: same as s2.b`

A **`Dash`** is a special type of **`Segment`**, shown with no end points.

`Circle` objects (and **`Rings`** and **`Disks`**) have two properties: one **`Point`** object, called **`c`** (center), and one **`double`** value, called **`r`** (radius, default value **`1`**). Of course, the center being a **`Point`**, a circle named **`f`** will host **`f.c.x`**, **`f.c.y`**, and **`f.r`**. Therefore, declarations can contain **`point`** objects and **`double`** values (when two points are declared as arguments, they determine a diameter: the center will be at midpoint, the radius will be half the distance between the two points):

Declaration	Resulting coordinates
`c1 = Circle()`	`0,0/radius 1`
`c2 = Circle (x,y,r)`	`x,y/radius r (type` *`double`*`)`
`c3 = Circle (p,r)`	`c3.c same as` *`point`* `p; radius c(`*`double`*`)`
`c4 = Circle (s)`	*`segment`* `s will be a diameter`
`c5 = Circle (p1,p2)`	*`points`* `p1 and p2 will define a` ***`diameter`***

A **`Disk`** is a special **`Circle`** with a filled interior.
A **`Ring`** is another special **`Circle`**, shown without the center.

`Line` objects have three properties: the coefficients **`a`**, **`b`**, and **`c`** of its first-degree equation. These, however, are not as easily "visible" as other data. So, it will be probably more common to declare a **`Line`** object from two **`Point`** objects (or a **`Segment`** object), or from four **`double`** values (corresponding to two points), or from two **`double`** values (corresponding to slope and y-intercept):

Declaration	Resulting line
`r1 = Line()`	`the x axis`
`r2 = Line(a,b,c)`	`ax + by + c = 0 (type` *`double`*`)`
`r3 = Line(m,q)`	`mx - y + q = 0, or y=mx+q (type` *`double`*`)`
`r4 = Line(a,b,c,d)`	`the line through (a,b) and (b,c) (type` *`double`*`)`
`r5 = Line(p1,p2)`	`the line through` *`points`* `p1 and p2`
`r6 = Line(s)`	`the line through` *`segment`* `s`
`r7 = Line(p)`	`the horizontal line through` *`point`* `p`

An **`Image`** object requires a file name (given as a **`string`** value), two **`double`** coordinates (**`x`** and **`y`**), and an optional scaling factor. As with **`Point`** objects, the default coordinates are **`(0,0)`**. **`Text`** objects are similar, but the string contains the value to be shown. Some examples:

Declaration	Resulting coordinates
`img1 = Image("mario1.bmp")`	`(0,0)`
`img2 = Image("mario2.bmp",x,y,0.5)`	`(x,y); 4th argument: scaling factor`
`img3 = Image(x,y,"mario3.bmp")`	`(x,y) (type` *`double`*`)`
`img4 = Image(x,"mario4.bmp",y)`	`(x,y)`
`txt1 = Text("Chicken on a raft",x,y)`	`(x,y)`
`txt1 = Text("the ocean blue",x,y,15,"Consolas")`	`(x,y), plus font name and/or size`

Summary of geometrical properties and methods

The following is a summary of the properties of geometric objects. In the case of those properties which are themselves objects, *their* properties are implicitly defined. Contrary to the declared objects, the names of the properties are fixed (being written in the designer class), and cannot be changed: for example, the centers of two circles named **`shoe`** and **`boot`** would both be named **`c`** (respectively, **`shoe.c`** and **`boot.c`**).

The properties of lines are the coefficients of the general equation (**`ax+by+c=0`**), which covers all cases, including vertical lines. However, since lines are often identified with the explicit equation (**`y=mx+q`**), these objects have been given additional methods for evaluating slope and intercept. Nothing really changes: with a little algebra, the explicit equation is shown to be equivalent to **`mx-y+q=0`** (so that **`a=m`**, **`b=-1`**, and **`c=q`**).

Object			Properties	other	type
`Point p`			`p.x`		`double`
			`p.y`		`double`
	`Pixel p`				
	`Image img`			`img.label`	`string`
		`Text t`		`t.label`	`string`
`Segment s`			`s.a`		`point`
				`s.a.x`	`double`
				`s.a.y`	`double`
			`s.b`		`point`
				`s.b,x`	`double`
				`s.b,y`	`double`
			`s.angle()`	`(method)`	`double`
			`s.slope()`	`(method)`	`double`
	`Dash d`	*End points not shown*			
`Circle c`			`c.c`		`point`
				`c.c.x`	`double`
				`c.c.y`	`double`
			`c.r`		`double`
	`Disk d`	*Filled*			
	`Ring d`	*Center not shown*			
`Line r`			`r.a`		`double`
			`r.b`		`double`
			`r.c`		`double`
			`r.slope()`	`(method)`	`double`
			`r.intercept()`	`(method)`	`double`
`Rectangle z`			`z.l`		`double`
			`z.t`		`double`
			`z.w`		`double`
			`z.h`		`double`
	`Frame z`	*Empty interior*			

The `Point` object has three subclasses, i.e. objects with the same properties (which makes it a *superclass*): in fact, they all can be geometrically associated to **`(x,y)`** coordinates. Of course, there are differences:

1. `Pixel` objects are always shown as single pixels (thickness **`1`**);
2. `Image` objects are used to display pictures (what else) on the viewport (the additional **`label`** property contains the *name* of the image file, such as `'Mario1.gif'`);
3. `Text` objects are used to display labels on the viewport (the additional **`label`** property contains the *string* value of the text); incidentally, `Text` is a subclass of `Image`: in this case, **`Image`** is called a *parent* class, and **`Point`** a *grandparent* class;

In **`Segment`** objects, the difference between **`angle()`** and **`slope()`** is that the former is the angle in radians between the end points **`a`** and **`b`**, while the latter is the slope of the corresponding **`Line`** object, which is, as we know from geometry, the *tangent* of the aforementioned angle.

A **`Dash`** is a special **`Segment`**; a **`Disk`** is a filled **`Circle`**; a **`Ring`** is an empty **`Circle`**; a **`Frame`** is an empty **`Rectangle`**. We will soon see examples of classes and subclasses.

How to use *Line* objects

As seen before, lines can be declared in several ways:

- from two points;
- from a segment;
- from 4 numbers (coordinates of two points);
- from 3 numbers (the 3 coefficients of the canonical form of the equation: `ax+by+c=0`);
- from 2 numbers (the 2 coefficients of the explicit form of the equation: `y=mx+q`); this is not possible for vertical lines, which have no slope and explicit equation.

If the definition is faulty (for example, when starting from two coincident points, or from null or zero coefficients), the curve defaults to a horizontal line:

```
s1 = Line(0,0,0,0)        #   x axis
s2 = Line(0,0,0)          #   x axis
s3 = Line(p)              #   horizontal line through point p
```

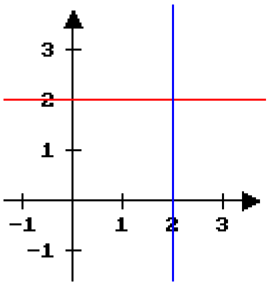
With no arguments, the curve defaults to the **x** axis:

```
s4 = Line()
```

Any line coincident with either axis can also be defined from predefined entities:

```
s5 = Line(xAxis)          #   s4 coincides with the x axis
s6 = Line(yAxis)          #   s5 coincides with the y axis
```

When we need lines parallel to the axes, there are a couple of ways to define them. In the examples, we will use the **red** horizontal line (`s1`) and the **blue** vertical one (`s2`), whose simplified equations are, respectively, `y=2` and `x=2`:



One algebraic method considers the coefficients of the equations, with the following reasoning:

Simplified equations	`y=2`	`x=2`
After transporting	`y-2=0`	`x+-2=0`
Canonical forms	`0x+y-2=0`	`x+0y-2=0`
Coefficients	`a=0`, `b=1`, `c=-2`	`a=1`, `b=0`, `c=-2`
Declarations	`s1 = Line(0,1,-2)`	`s2 = Line(1,0,-2)`

One geometric method uses the coordinates of two slack points (defined near the axes):

Choice of points	Horizontally aligned points	Vertically aligned points
Coordinates chosen	`(0,2)-(1,2)`	`(2,0)-(2,1)`
Declarations	`s1 = Line(0,2,1,2)`	`s2 = line (2,0,2,1)`

In many cases, only two arguments (slope and intercept) are needed:

```
s1 = Line(1,2)       #    slope=1; intercept=2
```

BEWARE !!! This cannot be done with *vertical* lines, which, in calculus, are somehow a pain in the neck, having *no defined slope*[11]. These lines may be defined with three or four numeric arguments, or with two vertically aligned points.

---

[11] The slope of a vertical line is *not* infinite, as some wrongly state. While it is true that the *limit* of the slope is ∞ as the line inclination approaches $\pi/2$, in a vertical line the slope just *does not exist*.

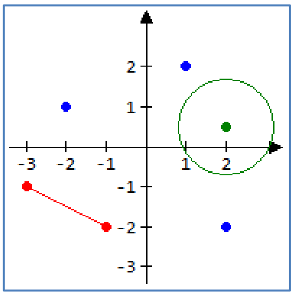
Understanding *thickness*

Visibility is fundamental in computer graphics[12]. For this reason, the point shapes of the graphic library have been designed with a default thickness of **3** pixels, making them small circles (actually, **3** is the *radius* of the circles, which have a diameter of **6** pixels).

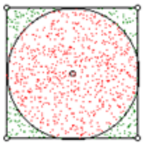
This works perfectly well when a small number of points is present in the viewport:

```
z = [Point(1,2), Point(2,-2), Point(-2,1)]
d.axes()
for x in z:
  d.draw(x,BLUE)
d.draw(Segment(-1,-2,-3,-1),RED)
d.draw(Circle(2,.5,1.2),GREEN)
```

Incidentally, the last two `draw` statements show how to sketch a shape "on the fly", without assigning it to any variable.



When many points are present, though, it is preferable to use `Pixel` shapes instead, as in the following simulation of a target hit by many green or red bullets:



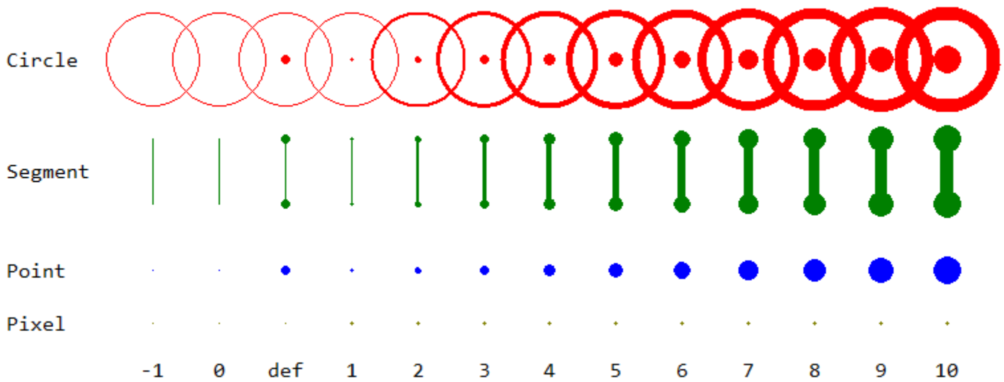
In order to provide a reasonable flexibility, all shapes can be drawn with a thickness of choice, expressed in pixels. The new argument can be placed either before or after the color, as in:

```
d.draw(x,GREEN,6)
x.draw(d,6,BLUE)
```

The rules are the following:

- by default, points are defined with a thickness of **3**, making them small circles **6** pixels wide;
- in a `draw` call, an integer value represents the brush thickness applied to the shape;
- points contained in certain shapes (segments and circles) have the same thickness of the parent shape;
- zero or negative values prevent the drawing of centers (in circles) and end points (in segments);
- such values, however, make `Point` and `Pixel` shapes collapse to **1** pixel;
- `Pixel` shapes can only be made a little larger with any thickness greater than zero;

The following image shows the effect of such calls with a variety of values (the `def` legend corresponds to the shapes drawn with the default thickness). Please note that `Pixel` shapes are very hard to see.



[12] Quite naturally, or there would *no point in it*.

Images

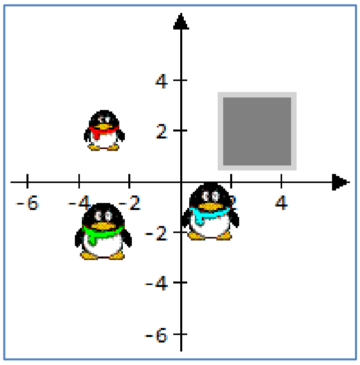
Images are not used in strictly geometric problems, but they are a staple of animations (which, after all, is geometry in motion).

In the following example, some **Image** objects will be stored in a *list*, a technique that will be used in future problems. There will be some scaling, and a reference to a wrong file name, which will generate a blank picture on the viewport. **Image** objects are normally centered around the chosen coordinates; in fact, **Image** being a *subclass*[13] of the **Point** class, it is treated in the same way, at least geometrically. It is still possible, though, to anchor an image in any legit mode (**NW,N,NE,W,CENTER,E,SW,S,SE**).

As the example shows, the arguments in the declaration of **Image** objects can be mixed, with the obvious rule that two numbers should accompany a string, in various orders. We will also take a look at some other shapes to show the difference with the regular objects.

```
g = Designer(200,200,-7,7,7,-7)
shapes = []                                  #   empty list
shapes.append( Image("mario1.gif",-3,2))     #   populate the list
shapes.append( Image("mario2.gif"))
shapes.append( Image(3,2,"wrong name.gif"))  #   such file does not exist
shapes.append( Image(-3,"mario3.gif",-2))
shapes[0].scale(0.7)                         #   image resizing (red scarf)
shapes[1].align('nw')                        #   alignment
shapes[2].anchor = 'w'                       #   equivalent alignment
for x in shapes:                             #   visit the collection
    g.draw(x)                                #   draw them all
g.show()
```

The image shows the result, with the wrong image (the file does not exist) producing a gray empty rectangle. We can observe the alignment of the second shape (cyan scarf) with respect to the origin: **'nw'** (top-left corner) refers to the position of the reference point (which is the origin), *not* to the image.



Note that the red-scarfed **mario1.gif** has been resized to **70%** of his original dimension.

NOTE: **Tkinter**, besides supporting only **GIF** image files (along with **PNG** files, but only in version **3**) does not support flipping and rotation. This is not a problem, though, for the purposes of this course.

Also, scaling is a little awkward, since it consists of a zoom in, followed by a zoom out, with *integer* factors: so the designer has to convert the scaling factor to a fraction, with a possible loss in quality.

Note for MAC users

On **MAC** systems, images must be referred to with the complete path name, as the following example shows. Of course, apart from **/Users** and the **image name**, the **rest** depends on how users organize their folders.

```
mario = Image('/Users/whoever/images/Mario1.gif',-3,-2)
```

In order to get the complete path name, the procedure is the following:

1) Open the folder where the images are stored;
2) Right click on any image file, then choose "obtain info"; alternatively, click **CMD + i**;
3) The path name will appear in a new window, from where it can be copied and pasted into the source code.

[13] This term denotes objects derived from a so-called *parent class* (in this case, the **Point** class), and may have different, or additional, properties and methods.

Text, and other subclasses

Text is not frequently used, but it will do no harm to see how it can be displayed in the viewport (for example, when labels or gauges must be shown). The example will look similar to the preceding one, with the use of a Python list.

Just like with **Image** objects, **Text** declarations can be mixed, with the obvious rule that two numbers should accompany a string, in various orders (after all, **Text** is a subclass of **Image**). **Text** objects also have a **newfont** method where font name, size and color can be changed (in any order). The default font is the monotype **Consolas**, size **12**. Regarding alignment, the **Text** class follows the same rules of **Image**, of which it is a subclass[14] , with the default anchoring being **'C'** (center). In case the alignment needed to be changed, the following method must be called (using any of the **Tkinter** anchoring constants):

```
t.align("S")
```

We will also take a look at the designer **clear()** method, which is not essential in the example (a new viewport is empty by definition), but shows how it is possible to clear the viewport and change the background color at the same time. In the example, the chosen color is **THISTLE** (=*cardo*). We will also take a look at some other shapes to show the difference with the regular objects.

A sample program could contain:

```
d.clear(THISTLE)                                   #   Clear with new color
shapes.append( Text(0,'chicken on a raft',5))      #   this is TEXT
shapes.append( Text(0,-5,'chicken on a raft'))
shapes[0].newfont(10,"BLUE",'Annone')
shapes[1].newfont(RED,14)                          #   TEXT characteristics
```

Font characteristics can also be changed individually, as in:

```
t.newfont(GREEN)
t.newfont(30)
t.newfont('Annone')
```

Speaking of subclasses, we can take a look at some more, namely:

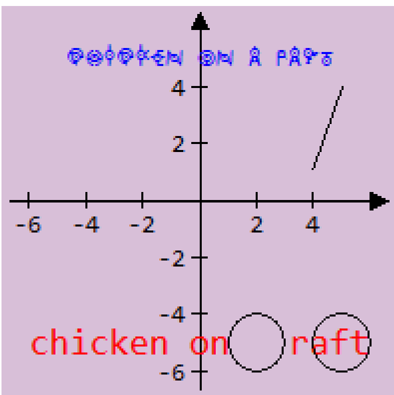
- **Disk** – a filled **Circle** (with two colors: border and interior);
- **Ring** – a **Circle** with the center not shown;
- **Dash** – a **Segment** with the end points not shown.

The **Ring** and **Dash** subclasses, just as with the **Pixel** object, can be used when the reference points need to be hidden. One use for the **Disk** shape in this course will be the simulate a pendulum bob, but many others could come in handy.

So we could complete our sample program with the following:

```
shapes.append( Disk(2,-5,1))
shapes.append( Ring(5,-5,1))
shapes.append( Dash(5,4,5,1))
```

Note that the **Disk** covers part of the **red** text (“chicken on a raft”), whereas the **Ring** does not.



WARNING: the chosen font must be regularly installed on the system, or the text will default to a rather stale **Arial**.

The same warning in ANNONE

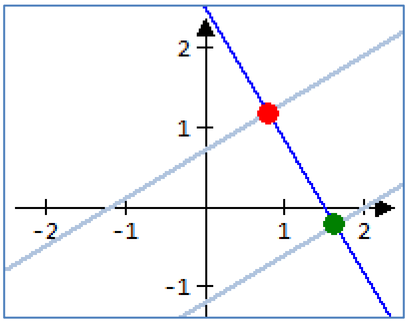
[14] Which makes **Point** a *grandparent* class.

Give us the tools, and we will finish the job[15]

Simply put, geometry is the interaction of shapes. Let us consider, for example, the notion of *distance*: in its basic concept, it is calculated between two points, but there are other distances possible, between: point and segment, point and line, point and circle, line and circle, and many others.

The distance between lines is a problem that can give the idea of the reasoning behind the geometric functions of the library. There are two cases: either the lines meet, or they do not (being parallel): in the first case, the distance is zero; in the latter, the distance is the fixed width of the strip cut out by the lines. This can be calculated by choosing a random point on one line, finding the perpendicular to the other line passing through that point, then finding the intersection between the two lines; the distance between the two points thus obtained is the desired result. This solution, however, requires a few steps for finding intermediate entities and/or values (the entities are described in the corresponding color):

- a point lying on a given line, given its abscissa (or ordinate, in the case of vertical lines);
- a line perpendicular to a given one, lying on a given point (in projective geometry parlance, tracing a line is called a *projection*);
- the intersection point between the two lines (also in projective geometry, a *section*);
- the distance between two points.



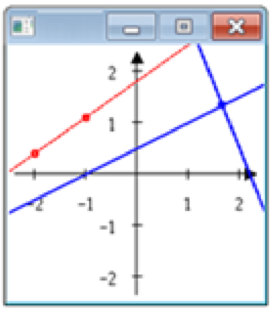
The first two steps require a little tinkering with the coefficients of lines; the third one requires Pythagoras's theorem. We can note, as a final consideration, that the problem always gives a solution (two perpendicular lines are guaranteed to meet).

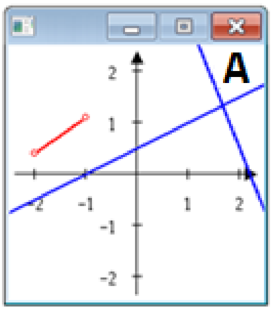
Anything missing? Of course, if the two lines coincide, the distance is again zero. But this case *is already covered*, since the result is reached anyway, with the small price of unnecessary calculations. Looking at the image, if the two black lines were coincident, the red and green points would also coincide, and the null distance would be calculated anyway; but this would only be a minor nuisance, and, what's more, the calculations are done by the program, not by us.

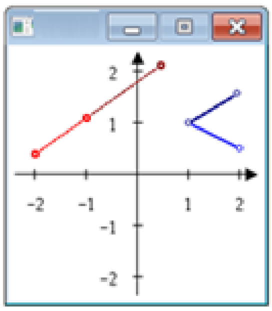
The basic tools

The majority of plane geometry problems can be solved with six elementary concepts, coupled by duality. However, there is a slight difference: while the duality for the first two concept regards, as customary, points and lines, in the other couples it regards ***distances*** and ***angles***.

- Two (distinct) points lie on one, and only one[16], *line*; two (distinct) lines lie on one, and only one, *point*;
- Between two points there is a *distance*; two lines meet at an *angle* (**A**);
- Segments can be expanded (or contracted) to a given *length*; they also can be rotated by a given *angle*.





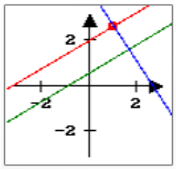


[15] Winston Churchill (February 9, 1941).

[16] This expression is not a whim of mathematicians: it is used because *"one"* and *"only one"* are two separate concepts.

Something useful

The objects defined in the designer are equipped with several **methods** for solving the basic plane geometry problems, like the example just shown. Putting together the hints, a proper program can be written.

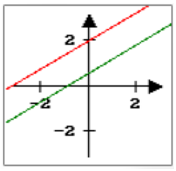


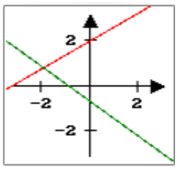
1 Finding the line parallel or perpendicular to another, passing through a given point:

```
r3 = p.parallel(r2)        #   returns a LINE object
r4 = p.perpendicular(r2)   #   returns a LINE object
```

2 Verifying if two lines are parallel (or perpendicular):

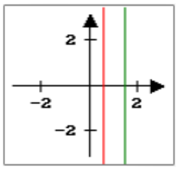
```
if r1.parallel(r2):        #   returns True/False
if r1.perpendicular(r2):   #   returns True/False
```

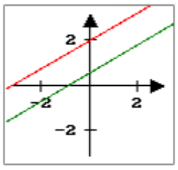




3 Verifying if a line is vertical (or horizontal):

```
if r1.vertical():          #   returns True/False
if r1.horizontalal():      #   returns True/False
```





4 Choosing a point on a line:

```
y = 1                      #   for non-horizontal lines
x = r1.abscissa(y)
x = 1                      #   for non-vertical lines
y = r1.ordinate(x)
```

5 Finding the point where two lines meet:

```
q = r3.section(r2)         #   the result is a Point object
```

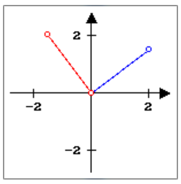
(more on `sections` later)

6 Evaluating the distance between two points (or a point and any other shape):

```
d = p.distance(q)
write(d)                   #   show on the shell window (console)
```

7 Rotating a segment by a given angle:

```
s.rotate(pi/2)             #   square (right) angle, on a
s.rotate(pi/2,1)           #   the same, but on s.b
```



The default call rotates the segment about its end point `a`. The second call (with an additional parameter) rotates the segment around its end point `b`.

The segment is shown **before** and **after** the rotation. This requires two `draw()`and at least one `show()`. Note that `pi` (complete name: `math.pi`) is a "magic constant" (contained in the Python `math` library) with the best approximation (at 15 digits) of the ratio of a half circumference to its diameter (**π**).

8 Any object can be initialized from another of the same type, assuming the same property values, but *not with a direct assignment*:

```
c2 = Circle(c1)         #   c1 is a Circle object
r2 = Line(r1)           #   r1 is a Line object
c3 = c1                 #   NOT LIKE THIS: c3 BECOMES ANOTHER INSTANCE OF c1
```

This is due to an (usually undesirable) feature called *structure sharing*: circle `c3` shares the same memory address as `c1`, making the two shapes one and the same. This way, any change made to either shape is replicated on the other (think of it as the same object having two names).

The dangers of structure sharing

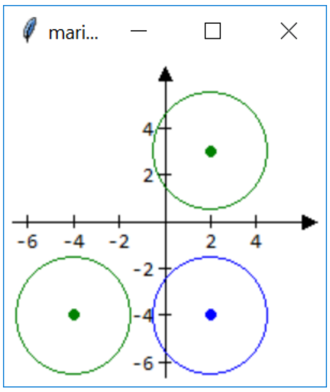
Careless programmers can easily fall prey to a sloppy declaration for an object cloned from another one.

We will see an example of what can happen when doing things improperly, by duplicating an entity the right way and the wrong way[17].

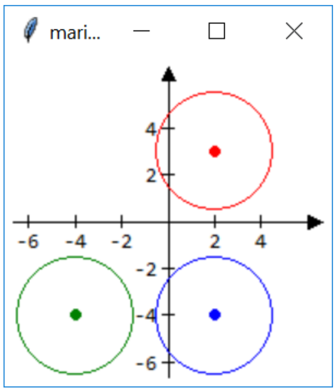
A circle named **a** will be declared. Two more will follow from it: circle **b** with no structure sharing (the right way), and circle **c** with structure sharing (the wrong way). Both **b** and **c** will be changed, but while the change to **b** will not affect **a**, the one on **c** will indeed cause circle **a** to change, because *they share the same memory address*. As mentioned before, it's like having two names for the same thing.

```
g = Designer(200,200,-7,7,7,-7)
g.axes(2,2)
a = Circle(2,-4,2.5)      #   The original circle
g.draw(a,BLUE)
''' Good example '''
b = Circle(a)             #   A new circle
b.c.x = -4                #   Change something
g.draw(b,GREEN)           #   Draw "b"
g.draw(a,BLUE)            #   No effects on "a"
''' Bad example '''
c = a                     #   "c" is the same as "a"
c.c.y = 3                 #   "a" changes along with "c"
g.draw(c,GREEN)
g.sleep(2)                #   wait a moment...
a.draw(g,RED)             #   "a" now covers "c"
''' Show in the shell '''
c.show()
a.show()
```

What happens is that any changes to **c** will also affect **a** (which is normally a bad thing):



A couple of seconds later…



After drawing **b** and **c**, circle **a** stays put

Circle **a** now covers circle **c**

The situation is also reflected in the shell, where a couple of calls to the **show()** method demonstrate how the coordinates of **a** have been changed along with those of **c**:

```
c(x,y) (2.000, 3.000 ); r : 2.5
c(x,y) (2.000, 3.000 ); r : 2.5
```

---

[17] Popular wisdom reports that "There is a right way, a wrong way, and the *army way*", to which some add: "which yields the same results as the wrong way, but takes much, much longer". This is not too kind on the military, but some may agree that waste of time and resources does indeed happen in public services.

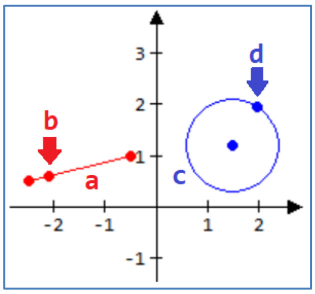
There are no indications about an "army way" for dealing with this problem. Anyone can decide on which side it would stand, if there were one.

Points and curves

Curves are made up by points, so the first thing that can be verified is whether a point lies on a given curve[18]. By the principle of duality[19], any such considerations can be reversed, so it is geometrically sound to state that if a point lies on a curve, then the curve lies on the point.

For this reason, the library contains several `on()` methods and functions (the name has been taken from the expression "a point lies *on* a curve"). These modules return `1` or `0`, since the relation between curves is either *true* or *false*. The functions are dual: using the jargon of projective geometry, a curve is considered *on* a point if the reverse is true. The `internal()` function determines the position of a point relative to a circle or a rectangle (`-1`, `0`, and `+1`, respectively for *inside*, *on*, and *outside*). The `inside()` function does the same, but returning only `True` (for *inside* or *on*) or `False` (for *outside*).

For example, let us assume that the **red** segment, the **red** point, the **blue** circle, and the **blue** point were called, respectively, `a`, `b`, `c`, and `d`, with these declarations:



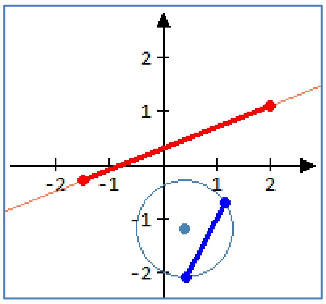

```
a = Segment(-0.5,1,-2.5,.5)
b = Point(-2.1,0.6)
c = Circle(1.5,1.2,0.9)
d = Point(c.c.x + cos(1),c.c.y + sin(1))
```

The following snippet produces the answers listed as comments to each line:

```
write(a.on(b))              #    True
write(b.on(a))              #    True (dual)
write(c.on(b))              #    False (b is not "on" c)
write(b.on(c))              #    False (dual)
write(a.on(a))              #    True (every curve is "on" itself)
write(b.on(b))              #    True
write(d.on(c))              #    True
write(c.on(d))              #    True (dual)
write(d.on(a))              #    False (d is not "on" a)
write(c.on(a))              #    False (c is not "on" a)
write(d.inside(c))          #    False (d is "on" c, not "inside")
write((c.c).inside(c))      #    True (c.c is "inside" c)
write(b.internal(c))        #    +1 (b is "outside" c)
```

The output of the `write` function appears, as usual, in the runtime window. It should be noted that said function is distributed with the designer, calling the appropriate Python `print` (which is a statement in version `2` and a function in version `3`, which are, for reasons unknown, totally incompatible).

A point can be tested to be "on" every other object. A segment can be "on" a point for duality, or "on" a line if both its vertices are (as the example shows with the **red** curves). Of course, it is hard to fit straight curves into circles, so the relation with circles is only valid for points and straight lines. Yet, with a leap of faith that could make prof. Barlotti turn over in his grave, a segment is considered "on" a circle if both its vertices are (as shown with the **blue** curves).



A list of methods and procedures, as well as a review of *trigonometry* (necessary to solve most geometric problems) is shown in Appendix B.

---

[18] Straight lines *are* curves. They just have an infinite radius of curvature (or a curvature equal to `zero`).

[19] Strictly speaking, duality refers to *points and lines* in *projective geometry*, an advanced branch of mathematics. Here, the term will be used in a broader sense, referring to other shapes as well.

A note on *polynomials* and related functions

The designer can be also used for problems in *calculus*, which is, simply put, the study of *functions*. For the sake of simplicity, only real-valued functions of a single real variable are treated (these entities can be represented with a **2D** curve). To begin with, the simplest of them are *polynomial functions*, such as:

$$y = x^4 - x^3 + 2x - 1$$

These functions are always continuous and derivable, and are easily studied. Furthermore, geometric processes can be also solved algebraically with simple computer programs[20], thus providing two related solutions, which can be compared to check the precision of a numerical method.

Declaration of a polynomial object can include the list of coefficients, ordered by the degree of the independent variable, as follows:

```
p1 = Polynomial()                 #    y = 0
p2 = Polynomial(4)                #    y = 4
p2 = Polynomial(-1,2,0,-1,1)      #    y = x^4 - x^3 + 2x – 1 (the example)
p2 = Polynomial(0,0,0,0,0,0,1)    #    y = x^6
```

Polynomials have the same display and sketch functions as the other shapes (`show()` and `draw()`):

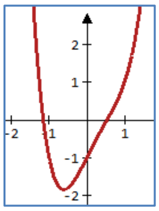
```
p = Polynomial(-1,2,0,-1,1)
p.show()                          #    in the execution window
p.draw(g)
g.draw(p,RED,3)                   #    color, thickness
p.draw(g,3,RED)                   #    the call is commutative
```

Result of `show()` in the shell window:

```
x ** 4 - x ** 3 + 2 x - 1
```

The powers of `x` are marked with a double asterisk, pursuant to the **Python** syntax (for the nostalgic, it is so in **COBOL**, too).

Result of `draw()` in the viewport:



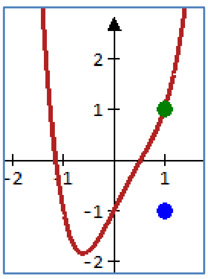
The `Polynomial` object has a `value` method (and an equivalent one called `y`) to compute the value of the function in the corresponding argument (for example, the abscissa of a `Point` object):

```
q = Point()
q.x = 1.2                         #    works as an independent variable
q.y = p.value(q.x)                #    function value = x^4 - x^3 + 2x – 1
q.y = p.y(q.x)                    #    EQUIVALENT
q.show()                          #    The point on the shell: (1.2,1.7456)
```

These are also a couple of `on()` methods, which verify if a given point lies on the curve corresponding to the polynomial (as usual, one belongs to the `point` object, the other to the `polynomial` object):

```
p1 = Point(1,p.y(1))
p2 = Point(1,-1)
write( p1.on(p))                  #    returns True
write( p.on(p2))                  #    returns False
```

Point `p1` (**green**) lies on the curve, having been constructed accordingly; point `p2` (**blue**) does not.



---

[20] The coefficients of a polynomial are stored into an *array* called **x**, which is a property of the **Polynomial** object.

Rational functions

Adding a degree of complication, we can extend the reasoning to rational functions, which are defined as algebraic fractions in the independent variable **x**, by using two polynomials (numerator and denominator, which are the properties of the corresponding object), as follows:

```
r1 = Rational((1,2,3),(1,2))     #    y = ( 3x^3 + 2x + 1 ) / ( 2x + 1 )
```

The default denominator is **1**, in which case the **Rational** object is equivalent to a **Polynomial** object[21]:

```
r2 = Rational((1,2,3),(1))       #    y = ( 3x^3 + 2x + 1 ) / 1
r3 = Rational((1,2,3))           #    Same result
```

A null denominator is not allowed, and if that were the case, it would again default to **1**:

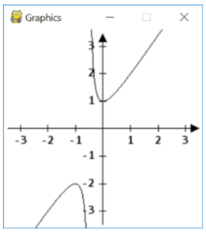
```
r4 = Rational((1,2,3),(0))       #    y = ( 3x^3 + 2x + 1 ) / 1
```

It is also possible, just for the fun of it, to use stray numbers at either end of the argument list), but it is not a suggested practice, as it would break the symmetry of the declaration:

```
r5 = Rational(1,2,3,(1,2))       #    y = ( 3x^2 + 2x + 1 ) / ( 2x + 1 )
r6 = Rational((1,2,3),1,2)       #    Same result
```

Other methods, like **on()** and **draw()**, work just like with polynomials:

```
r1.draw(d)                       #    where "d" is a Designer object
d.draw(r1)                       #    commutative call
```



Generic functions

Python (unlike **C/C++**, but like some more modern languages) makes it possible to define any function with a mathematical formula stored into a *string*. There are two rules to follow:

- It must be a function in **x**;
- the formula must be mathematically correct (NOTE: the power operator in Python is ******, not **^** as some insist on writing, thinking they are using a spreadsheet).

Some examples:

```
f1 = Function("sin(x)")          #    Correct
f2 = Function("sin(z)")          #    Wrong (not in x)
f3 = Function("sun(x)")          #    Wrong (wrong function name)
f5 = Function("sin(x**2)")       #    Correct (x^2)
f6 = Function("sin(x^2)")        #    Wrong (wrong operator for x^2)
```

With wrongly declared functions, the dependent variable is always evaluated to **None**:

```
y = f1.y(2)                      #    Evaluates to 0.9092974268256817
y = f3.y(2)                      #    Evaluates to None
```

Of course, **draw** and **on** methods are present.

These objects rely on the flexibility of Python, which, with the aptly named **eval()** function, can *evaluate* "on the fly" any string containing a mathematical expression, provided its syntax is correct and it refers to defined variables[22]. This is easily verifiable from the command prompt, as in the following example:

```
>>> a=pi
>>> eval("cos(a)")
-1.0
```

---

[21] Equivalent does not mean "the same thing": it still remains a **Rational** object, not a **Polynomial**.

[22] The reason why **Function** objects require an expression in **x** is that the evaluation methods use a parameter by that exact name (a design choice made to comply with the accepted practices of calculus).

Projections and sections

Victor Poncelet
(1788-1867)

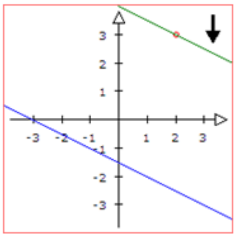
This is a journey into the storied world of Euclidean geometry, where parallel lines do not meet[23]. Some concepts and terms, however, will be taken from *projective* geometry (a non-metric geometry based on points and lines, developed mostly after the work of the French mathematician Victor Poncelet), the main ones being:

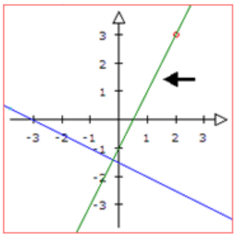
- A *projection* is drawing a line from a given point (or, more generally, from a given shape, where a number of lines can be drawn).
- A *section* is when two curves (or shapes) meet.

Parallel and perpendicular lines are typical projections. As Euclid's Fifth Postulate states, given a line and a point outside it, there is one, and only one, line through that point that is parallel to the given line[24]. Actually, for practical purposes the point could also be *on* the line, in which case the resulting line would be coincident with the original one; the related problem of finding a perpendicular line has no such limitations.

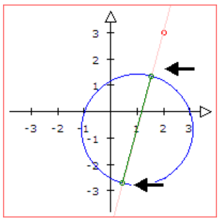
Theoretical considerations apart, with a point **p** and a line **r**, we can obtain another line **s**, as follows:

```
s = p.parallel(r)
s = r.parallel(p)          #    the call is dual
s = p.perpendicular(r)
```





A point can be projected towards any shape, determining, where possible, another point on said shape (in fact, a perpendicular to a given line sections the line in a point called the *orthogonal projection* of the point, or, less informally, the *foot* of the point). So the concept of projection is extended with the section of the projected line with the other shape. The notable exception is the circle, where the result is a segment (as shown in the image to the right)[25].

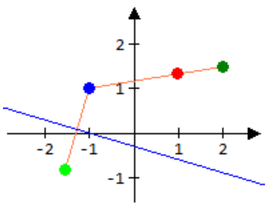


For example, considering points **p** and **q**, a segment **s**, a line **r**, and a circle **c**, we may write:

```
p1 = p.projection(q)      #    point
p2 = p.projection(r)      #    point
p3 = p.projection(s)      #    point
s2 = p.projection(c)      #    segment (a diameter)
```

The projection of a point can be extended with a number **d**: in this case, the result is another point, lying on the line joining the two given shapes, at a distance of **d** from the *first*, going towards the *second* (or in the opposite direction, if a *negative* distance is used). In the first example, the **red** point **z1** is obtained by projecting the **blue** point **p** towards the **green** point **q** by a distance of **2**; in the second one, the **lime** point **z2**, symmetrical of point **p** with respect to line **r**, is obtained by doubling the distance **p/r**:

```
z1 = p.projection(q,2)
z2 = p.projection(r,2 * p.distance(r))
```



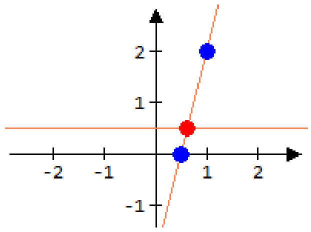
[23] There are geometries where the concept of parallelism is overturned, when not outright abandoned.

[24] Strictly speaking, this is an *equivalent* postulate, stated by the Scottish mathematician John Playfair (1748-1819). Euclid's wording is a bit more complicated.

[25] Circles are *second-degree* curves, so two points can be expected when sectioning them with lines and other circles..

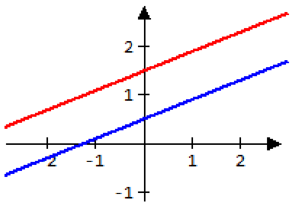
A *section* occurs when shapes meet, the most conspicuous case being with two *lines* (when they are distinct and not parallel). In order to follow the duality principle, a section between two *points* is the line joining them (when they are distinct). In the following example, lines `z1` and `z3` coincide, having only been obtained with different calls (the former to the `line` constructor, the latter to the `section` method)

```
p1 = Point(1,2)          #   two points at will (blue)
p2 = Point(0.5,0)
z1 = Line(p1,p2)         #   line joining the two points (coral)
z2 = Line(.5)            #   parallel to the x axis
p3 = z1.section(z2)      #   point joining two lines z1 and z2 (red)
z3 = p1.section(p2)      #   The same as z1, calling a method
```



Of course, there are cases when curves do not meet (for example, parallel lines, segments too short, divergent rays). When this happens, the library returns an object where the values of the numeric components is set to `nan` (not a number), which can be detected with a call to the `isnone()` method.

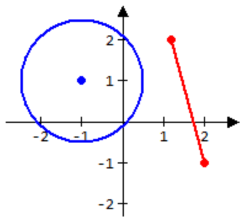
```
r1 = Line(.4,0.5)
r2 = Line(.4,1.5)
r1.draw(g,BLUE,2)
r2.draw(g,RED,2)
p = r1.section(r2)
p.draw(g,GREEN,3)       #    NOTHING is drawn
p.show()                #    nan
print(p.isnone())       #    True, in this case
```




```
x,y (nan, nan)
True
```

In the following snippet, the **circle** and the **segment** are external; their section, the hypothetical segment `s`, is nowhere to be seen in the viewport, and its properties can only be shown in the shell, as well as the result of the call to `isnone()`:

```
c1 = Circle(-1,1,1.5)
c2 = Segment(2,-1,1.2,2)
c1.draw(g,BLUE,2)
c2.draw(g,RED,2)
s = c1.section(c2)
s.draw(g,GREEN,3)       #    NOTHING is drawn
s.show()                #    nan
print(s.isnone())       #    True, in this case
```



The coordinates of the end points of the segment make sure that no processing can be done on segment `s`: any attempt would result in equally empty result.

```
a(x,y),b(x,y) ( nan , nan );( nan , nan )
True
```

If we tried to project the origin onto the empty segment `s`, the resulting "point" `z` would be another empty, and consequently invisible, shape:

```
z = O.projection(s)
z.show()
```

> It is not necessary to check the validity of `s`: the result `z` will be just another null shape.

```
x,y ( nan , nan )
```

So, there is nothing to worry about such cases: the graphic library takes care of everything, and any calculations involving the "phantom" shapes will only yield more phantom shapes, without raising errors.

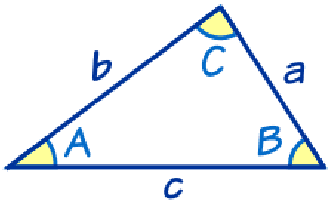
Solving triangles: a summary

Often, when solving a problem with geometric means, triangles come in handy. Here is a brief reminder on the topic. The convention is to call sides with lowercase letters, and the opposite angles with the corresponding uppercase letters.

Please note that some of the following are formulas, not programming language statements.

Sum of angles

```
A + B + C = pi
```



Laws of sines

```
a / sin(A) = b / sin(B) = c / sin(C)
```

Laws of cosines

$$\cos(A) = (b^2 + c^2 - a^2) / 2bc$$

Solving a triangle means finding the unknown elements, given some others (usually 3, at least one of which must be a side). The most common cases are listed here. Note that in the SSS and SAS cases, the given elements are listed: for other choice of letters (for example, **b, c, and A**), the solution must be changed accordingly.

SSS (given 3 sides: a, b, and c)

From the law of cosines:

$$a^2 = (b^2 + c^2 - 2*b*c*\cos(A))$$
$$A = \mathrm{acos}((b^2 + c^2 - a^2) / 2*b*c)$$

Hipparchus of Nicaea
(c. 190-120 BCE)

SAS (given 2 sides and the angle in between: a, b, and C)

Use law of cosines:

$$c = (a^2 + b^2 - 2*a*b*\cos(C)) ** 0.5$$
$$A = \mathrm{acos}((b^2 + c^2 - a^2) / 2bc)$$
$$B = \mathrm{acos}((a^2 + c^2 - b^2) / 2ac)$$
$$C = \mathrm{acos}((a^2 + b^2 - c^2) / 2ab)$$

Aryabhata
(476-550 CE)

SSA (given 2 sides and the angle NOT in between: a, b, and A)

First compute `D = b / a * sin(A)`

If `D>1` the triangle is not solvable (not a triangle);

If `D==1` the triangle has a square angle in `B` (then `C = PI - A - B`);

If `D<1`, there could be `1` or `2` solutions

First solution:

```
B = asin(D)
C = PI - A - B
```

Regiomontanus
(Johannes Müller von Königsberg)
(1436-1476 CE)

When `C` is found, use the laws of sines:

```
c = a * sin(C) / sin(A)
b = a * sin(B) / sin(A)
```

The second (possible) solution for `B` is `PI-D`, which can be verified by checking the sum of the angles (the solution would not be acceptable if the sum exceeded `PI`).

ASA, AAS

In either case, compute the third angle:

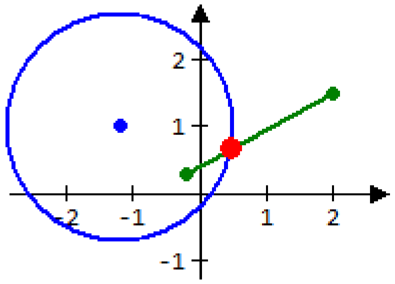
```
C = PI - A - B
```

then use the laws of sines.

Leonhard Euler
(1707-1783 CE)

Case study: a degenerate segment

Nothing immoral or perverted here: degenerate shapes exist when they lose their usual appearance. The case under examination is a segment-circle section where the shapes meet in one, and only one, point:



For the sake of consistency, the solution given in the geometric library is to still yield a segment, but a collapsed one, obtained by assigning the same coordinates to both end points. Its length will, of course, be zero. The following snippet shows said length in the shell:

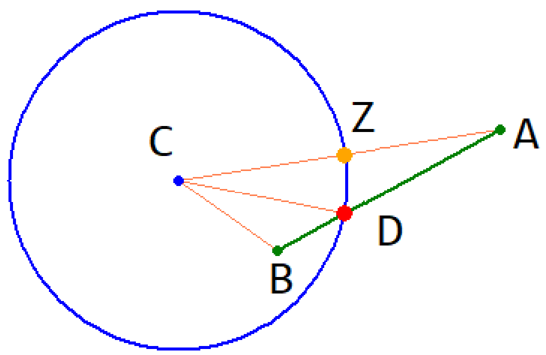
```
c = Circle -1.2,1,1.7)
s1 = Segment(2,1.5,-.2,0.3)
c.draw(g,BLUE,2)
s1.draw(g,GREEN,2)
s2 = c.section(s1)
s2.draw(g,RED,5)                # looks like a point, but it's a segment
print(s2.length())              # show length in the shell
```

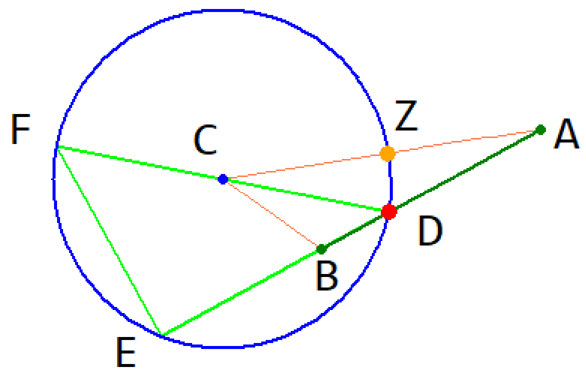
For the solution to the circle-segment section, the library uses the line where the segment lies as a prop. If said line intersects the circle in a new segment **s2**, the end points of segment **s2** are matched to the original segment, to determine their acceptability.

A purely geometric solution (using triangles) exists, quite elegant but also complex: it is shown here just for reference, with the construction in the image below. As in the previous snippet, the original shapes are the **blue** circle centered in **C** and the **green** segment **AB**. The steps of the solution are the following:

1. triangle **ABC** is a SSS case, so it can be solved;
2. triangle **BCD** is a SSA case where the angle is not between the sides (angle: **DBC**; sides: **BC** and **CD**), so the law of sines can be used (in such cases, the solution is normally not unique, since two angles exist with the same sine: **x** and **pi-x**, one acute and one obtuse);
3. find angle **BDC** opposite **BC** by the law of sines: this angle can only be acute, since angle **DEF**, obtained by extending **AB** and **DC**, is right (so the obtuse solution can be discarded);
4. find angle **ACD** by subtraction (**pi - ABC - BDC**);
5. find angle **BCD** by subtraction (**ACB - BCD**);
6. rotate radius **CZ** by **BCD** (**Z** is the **orange** point, easily obtained by projection from **C**);
7. it remains to be seen if the last rotation should be positive or negative (that depends on the relative positions of **AC** and **AB**): this can be verified by checking **D** against the original segment **AB**.

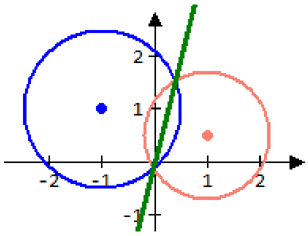
It should be noted that the solution uses triangles with point **D** as a vertex, without actually *knowing* its position (only that certain properties are satisfied). This trick can be useful in other situations, as well.



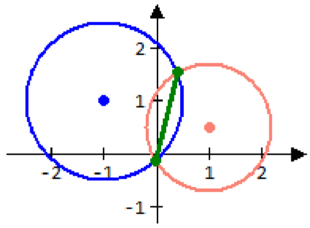


Case study: when circles meet

As an example of a geometric solution, let us consider the section between two circles, `c1` and `c2`, which, under normal circumstances, is a segment. Of course, said segment lies on the line called *radical axis*



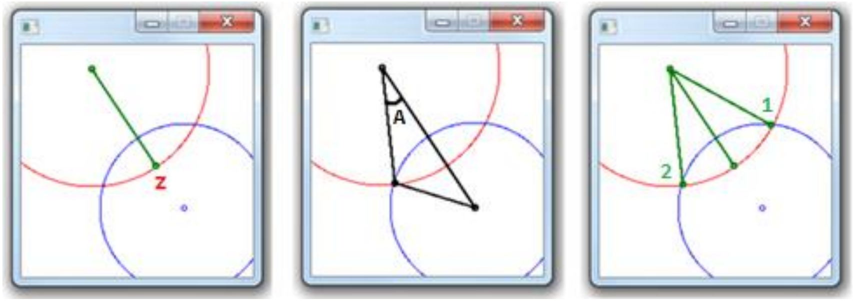
The radical axis



The section

The problem can be solved with the following steps:

- Verify if the two curves meet (by geometric, not algebraic considerations: it is simpler to verify if the distance between the two centers (calculated with the `distance()` method of the geometric library) is smaller than the sums of the two radii, rather than mess with the equations);
- Find the radius segment `s` of one circle, pointing towards the center of the other (find point `Z` first, then determine the **green** segment);
- Solve the (**black**) triangle joining the two centers and any of the two end points of the desired section (so as to find the outlined angle, which we can call `A`);
- Rotate the radius twice: first by the angle `A`, then by `-2*A`; the far end points found after each rotation define the resulting section between the two circles.



The angle for the final rotations can be found because the three sides of the black triangle are known; as usual, the sides are indicated by lowercase letters (`a`, `b`, and `c`), and the opposing angles by capital letters (`A`, `B`, and `C`), and the angle can be found by the law of cosines (see previous page for more on triangles):

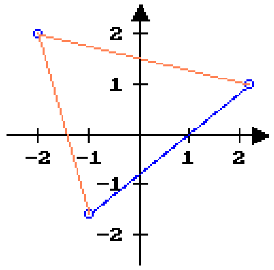
$$A = acos((b^2 + c^2 - a^2) / 2*b*c)$$

```
def section(c1, c2):
    q = Segment(None)
    if isinstance(c1,Circle) and isinstance(c2,Circle):
        if c1.c.distance(c2.c) <= c1.r + c2.r:
            z = (c1.c).projection(c2.c, c1.r)    #    point on the circle
            s = Segment(c1.c, z)                 #    the radius
            a = c2.r                             #    sides of the triangle
            b = c1.c.distance(c2.c)
            c = c1.r
            A = acos( ( b**2 + c**2 - a**2 ) / ( 2 * b * c ) )
            s.rotate(A)                          #    rotate once
            q.a = Point(s.b)                     #    first end point
            s.rotate(- 2 * A)                    #    rotate opposite
            q.b = Point(s.b)                     #    second end point
    return q
```

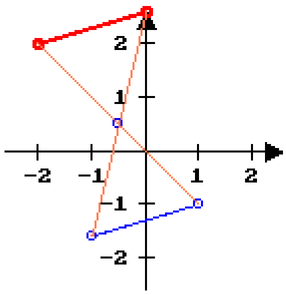
It should be noted that this function exercises extreme care in handling the parameters, verifying if they are indeed `Circle` objects with the `isinstance` function. This is not strictly necessary, for those who know what they are doing. Also, a null segment is defined at the beginning, to ensure that a `Segment` object is returned in any case.

Supplementary problems

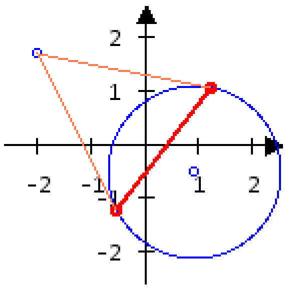
1 Given a point **p** and a segment **s**, find the area of the triangle obtained by **joining** point **p** to the two ends of segment **s**.



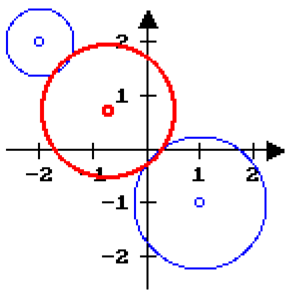
2 Given a point **p** and a segment **s**, construct the **symmetrical segment** of **s** with respect to **p**.



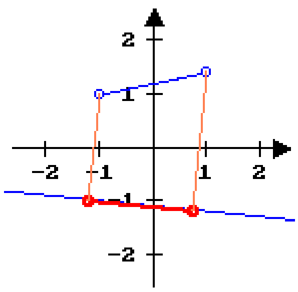
3 Given a circle **c** and a point **p** (outside **c**), find where the two tangents from **p** to **c** meet the circle.



4 Given two circles **c1** and **c2** (external with respect to each other), find the **smallest circle** tangent to both and centered on the same line as **c1.c** and **c2.c**. There are four such circles, which can be found for further practice.

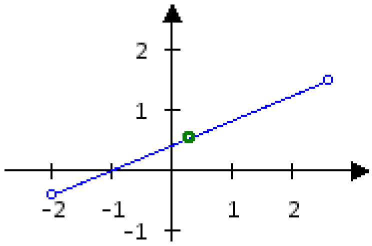


5 Given a segment **s** and a line **r** (with no points in common), find the **area of the trapezoid** (Brit.: **trapeze**) obtained projecting the end points of the segment onto the line.

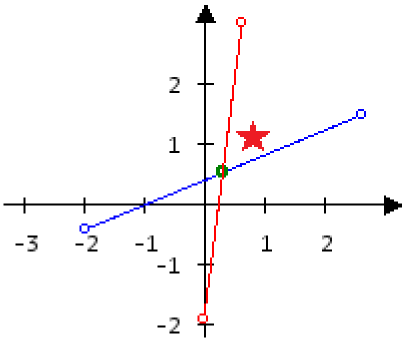


In the following problems, using a *function* can bring the solution one step further.

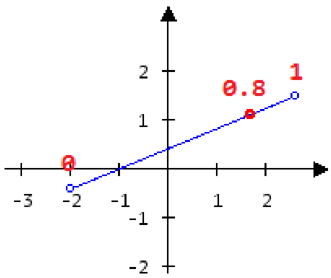
6 Given a segment `s`, find its midpoint.



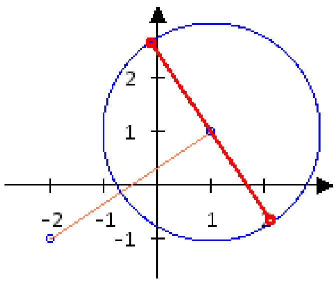
7 Given a segment `s` and an angle `a`, spin the segment on its midpoint by said angle.



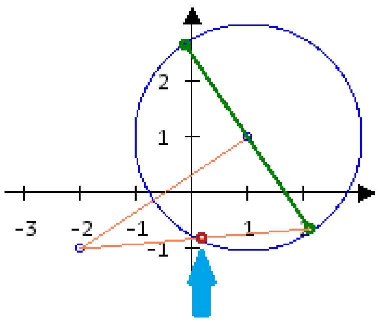
8 Given a segment `s`, and a number `x` ∈ `[0,1]`, find the point on the segment corresponding to the parametric coordinate `x` (`s.a` corresponds to `0`, and `s.b` to `1`).



9 Given a circle `c` and a point `p` outside `c`, join `p` with `c.c` and find the diameter perpendicular to that line.



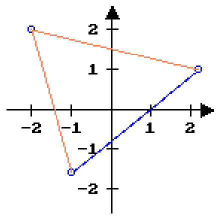
10 Proceed joining `p` with one end point of said diameter, then find the other point where the segment meets the circle.



Solutions

1 Find the two remaining segments of the construction (`p−s.a` and `p−s.b`), then use Heron's formula with the three segment lengths (in the formula, `p` is the semiperimeter of the triangle). Heron (or Hero) of Alexandria was a Greek mathematician who lived in the first century AD.

$$A = \sqrt{p(p-a)(p-b)(p-c)}$$

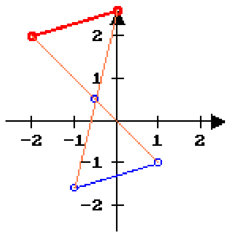


Another solution can be found by obtaining the line passing through segment `s`, then calculating the distance from `p` to said line (the height relative to base `s`) with the familiar formula `b*h/2`:

```
r1 = Line(s)
d = p.distance(r1)
```

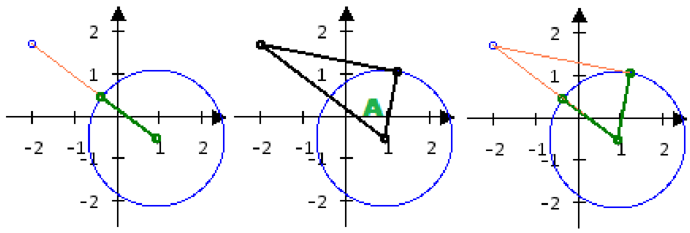
2 To find one of the end points of the new segment, project `s.a` towards `p` by the double of their distance:

```
s1.a = s.a.projection(p,2*p.distance(s.a))
```



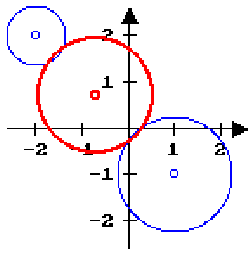
3 The **black** triangles formed by `c.c`, `p`, and either of the two points of tangency are square. Angle `A` can be found with the inverse cosine formula. It can then be used to rotate the **green radius** (obtained projecting `c.c` towards `p` by `c.r`) twice, in *opposite directions*. At each rotation, point `b` of the green radius coincides with one of the points of tangency. The second rotation must be `-2*A`, to compensate for the first one.

```
q = c.c.projection(p,c.r)
s1 = Segment(c.c,q)
s2 = Segment()
A = acos(c.r/p.distance(c.c))
s1.rotate(A)
s2.a = Point(s1.b)
s1.rotate(-2*A)
s2.b = Point(s1.b)
```



4 The radius `c3.r` of the new circle is the distance between `c1.c` and `c2.c`, minus the two radii (`c1.r+c2.r`), and divided by `2`. Projecting `c1.c` towards `c2.c` by `c1.r+c3.r` yields the center of the new circle.

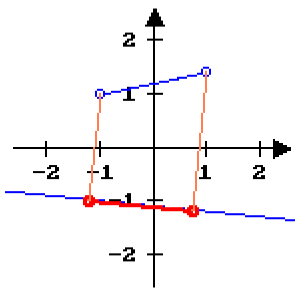
```
c3.r = ( c1.c.distance(c2.c) - c1.r - c2.r ) / 2.
c3.c = c1.c.projection(c2.c,c1.r+d)
```



Similar considerations apply to the other circles tangent to both `c1` and `c2` (of which there are three).

5 The projection of segment `s` can be obtained by projecting each of its end points onto line `r`. Let us call it `s2`: being perpendicular to said line, it can be considered the height of the resulting trapezoid. The two bases of said trapezoid can be calculated by using the `distance()` method, and the height by using the `length()` method.

```
s2.a = s.a.projection(r)
s2.b = s.b.projection(r)
b1 = s2.a.distance(s.a)
b2 = s2.b.distance(s.b)
A = ( b1 + b2 ) * s2.length() / 2
```



6 The function should return a point, so the header should look like:

```
def midpoint(s):
```

In the body of the function, the new point can be obtained in two different ways:

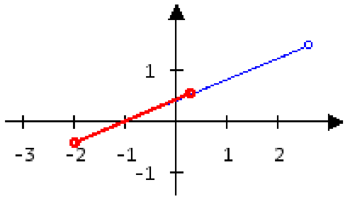
Algebraic solution	Geometric solution
`new.x = ( s.a.x + s.b.x ) / 2.` `new.y = ( s.a.y + s.b.y ) / 2.`	`new = s.a.projection(s.b, s.length()/2)`

This is one of those rare instances where the properties of the new shape can be calculated explicitly.

7 The function modifies the shape, so it should be a **`void`** function with a reference by address:

```
def spin(s,a):
```

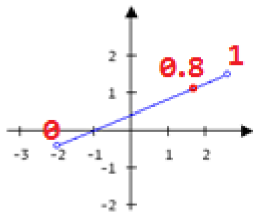
By using the midpoint (from the preceding problem), it is possible to determine a *half segment*, as it were, which can be rotated by the given angle, yielding one end point of the modified segment. The other end point can be obtained either with another rotation, or with a projection through the midpoint.



8 The function should return a **`Point`** shape:

```
def at(s,d):
```

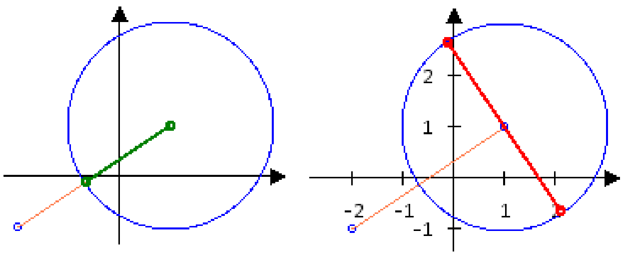
Here, the distance parameter could be checked against the interval **`[0,1]`**:




```
if d<0:
    d=0
elif d>1:
    d=1
```

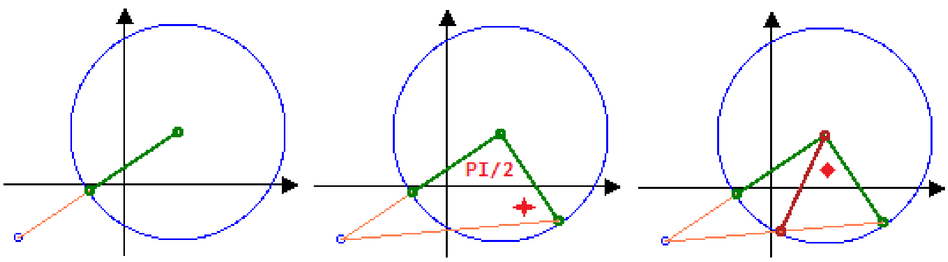
The desired point can be found with a projection, multiplying the length of the segment by the parameter.

9 The solution is typical of problems with circles. First, we can determine a slack radius (projecting **`c.c`** towards **`p`**); with each of two rotations (one by **`PI/2`**, the other by **`PI`**), the far point of the segment corresponds to the end points of the diameter.



1 - slack segment   2 - after two rotations

10 The solution can be independent from the preceding one. First, we can determine the usual slack radius (projecting **`c.c`** towards **`p`**); with a rotation by **`PI/2`**, a right triangle is formed, which can be solved for the angle marked with a *star*. Another isosceles triangle (where the newly found angle is at the base) can be solved for the angle marked with a *diamond*, for the final rotation of the slack segment.



1 - slack segment   2 - first rotation   3 - second rotation

Curve tracing

with geometrical methods

Geometry Systems and simulations

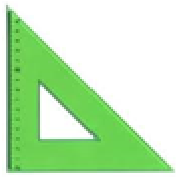
Drawing curves

Geometers have discovered many methods for tracing curves on a plane surface, building the corresponding physical devices, varying from the very simple (straightedge and compass, shades of Euclidean geometry) to the utterly bizarre:

Straightedge

Compass, protractor

Compass



Set square

Three-pointed compass

Ellipser

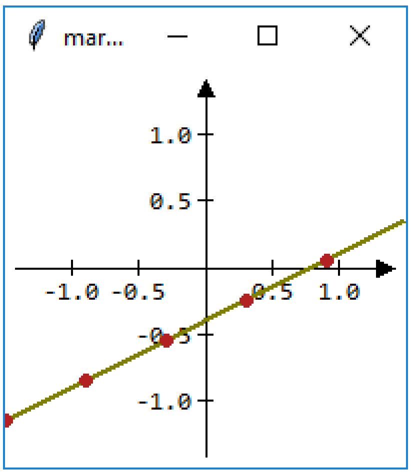
Compass for hyperbolas

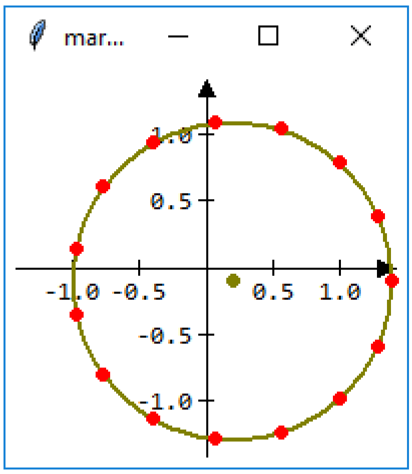
Conchoid guide

Once a programming language/library can manage a graphic window, it is possible to simulate an ideal "pen" for tracing any curve, given its geometric characteristics, using the same movement allowed by a drawing instrument. The key concept here is taking a point and *moving* it along a predefined curve, which, in the most basic cases, is either a line or a circle. These two correspond to straightedge and compass, but with the advantage of *graduation* over our Greek forefathers (that is, the possibility of measuring distances and arcs, which they did not allow in Euclidean geometry). Tracing a moving point on a viewport can be assimilated to the act of moving a pen or pencil against a paper sheet: of course this action requires a loop, inside which the coordinates of the moving point will change following some rules of geometry.

Such a curve is called a *pedal* line (from *podòs* or **ποδόσ**, the Greek word for *foot*; in fact, some call it *podal*, as in Italian, where the expression is *linea podale*). For each point of the pedal, other points can be found and drawn, following a geometric rule: the result is a new curve called a *kinematic curve* (from *kinein* or **κινειν**, the Greek verb meaning *to move*[26]). Besides a bit of geometry, the management of kinematic curves requires some special features of the designer, which will be introduced shortly.

But first it is necessary to analyze how to travel along a pedal line. The simplest one is arguably the straight line, which is normally not vertical (so as to comply with the common treatment of calculus). This way, a point can be moved along said curve by changing its abscissa and evaluating a new ordinate as well. Depending on how far away the points are from one another, the result can simulate a solid line (**green**), or appear as a sequence of points (**red**). With similar considerations (and some trigonometry), analogous results can be obtained on a circle.





[26] Not to be mistaken for the "polka *a chinein*" from Bologna.

Moving along pedal lines

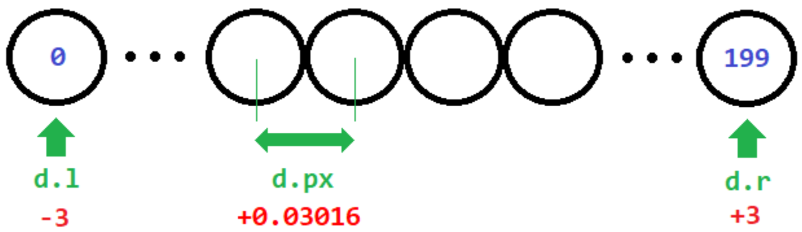
Before embarking on a trip along any curve, it is necessary to understand how to refer to the viewport, especially with regard to the boundaries.

As an example, a **200x230** viewport is calibrated against a **-3/3/3/-3** square world, so that **6** world units correspond to **200** horizontal pixels and **230** vertical pixels. After calibrations, the designer (called **d** in this example) will have the following properties:

	Properties	World	Screen	
Left margin of the world	`d.l`	`-3`	`0`	**WARNING !!!** This is only an example: the resulting values depend on the calibration. Do not make the mistake of thinking that one pixel is *always* 0.03 or 0.02 units.
Right margin of the world	`d.r`	`+3`	`199`	
Bottom margin of the world	`d.b`	`-3`	`199`	
Top margin of the world	`d.t`	`+3`	`0`	
World size of a *horizontal* pixel	`d.px`	`+0.03016`	`1`	
World size of a *vertical* pixel	`d.py`	`+0.02620`	`1`	

Calibration matches **screen coordinates** to **world coordinates** through rather simple arithmetic proportions; some **designer properties** allow us to span the viewport without any considerations for the actual numeric values. Keep in mind, however, that this is *only an example*: different calibrations will yield different values.

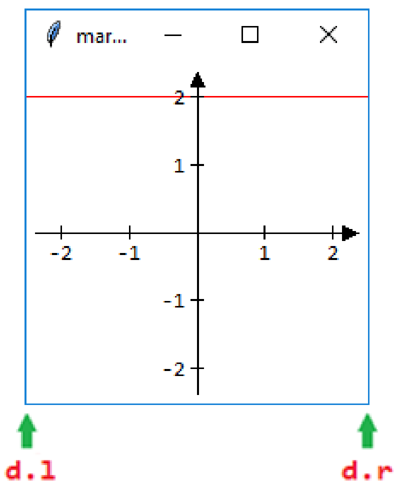
For example, a typical pedal is a horizontal line, spanning the whole viewport, one pixel at a time, thus covering all the screen window. Clearly, we need a loop: the good thing is that the boundaries and the step of the loop need no more than 3 designer properties. If we imagine the pixels enormously enlarged, the situation can be summarized in the following image:



We will start with the simplest pedal, a horizontal straight line (for example, **y=2**): from previous geometry courses, we should know that we need a **Point** object with ordinate **2** and a variable abscissa, spanning from **d.l** to **d.r**, with a step of **d.px**.

A point moving on this pedal will keep ordinate **2**, as shown in the following snippet.

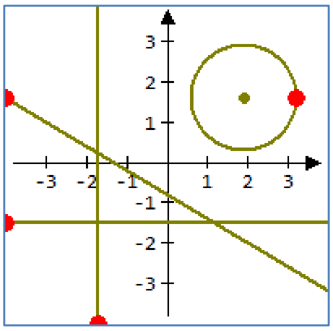
```
p = Pixel(0,2)      #   A single pixel, not a Point
p.x = d.l           #   Start
while p.x<=d.r:     #   Check for end of loop
    p.draw(d,RED)   #   Trace another point
    p.x+= d.px      #   Move by one world pixel
```



The main concepts are the following:

- The shape called *pixel*, as the name suggests, can be used for a single pixel of the graphic window, in cases where a large number of points is required (many points drawn as circles clutter the viewport);
- The properties **d.l** and **d.r**, after calibration, contain, respectively, the left and right margin of the *world* window (**d.t** and **d.b** for the top and bottom ones);
- Property **d.px** contains the world distance corresponding to a horizontal pixel (a different property, **d.py**, is used for vertical pixels, since the calibration may not be isometric);
- There is no need to draw the actual line, which will be traced by point **p**, simulating a pen.
- Simply put, the loop just shown moves point **p** along the pedal line **y=2**, with a step corresponding to one pixel.

Other lines



Different pedal curves

Depending on the problem, other pedal lines are possible. The simplest cases are non-horizontal lines and circles. Any curve is a candidate pedal, but we not able to draw them yet, so we will settle with those.

Vertical lines

A vertical pedal line can be traversed in the similar way as a horizontal one, substituting **g.b**, **g.t**, and **g.py** in the loop, and keeping the *abscissa* of the point constant.

Oblique lines

For an oblique line **r**, the suggested method is to traverse the world window from left to right, then obtaining the ordinate of the point with the corresponding method of the *Line* object:

```
p = Pixel(0,2)
p.x=g.l
while p.x<=g.r:
    p.y = r.ordinate(p.x)
    ...
    p.x+=g.px
```

The calculation of the ordinate could be done with horizontal lines as well: with a small waste of processor time, the advantage would be using the same loop structure.

Circles

When the pedal line is circular, point **P** moves accordingly; in this case, one solution is using a segment **s**, linking the center of the circle with **P**, starting from a horizontal position, then rotating it from **0** to **2*pi**.

```
a = 0
s = Segment(c.c, Point(c.c.x+c.r,c.c.y))
da = 0.01
while a <= 2 * pi:
    ...
    s.rotate(da)
    p = Pixel(s.b)
    a+= da
```

Another solution moves point **P** with the classic goniometric functions (*sine* and *cosine*). As we all know, the cosine is associated to the abscissa, and the sine with the ordinate of the moving point:

```
a = 0;
while a <= 2*pi:
    p.x = c.c.x + c.r * cos(a)
    p.y = c.c.y + c.r * sin(a)
    ...
    a+= 0.01
```

NOTE

A rotation of **0.01** radians corresponds to less than **1** sexagesimal degree, which is normally good enough. Of course, the rotation can be adjusted if the rendering is not satisfactory.

Segments

The problem with segments is that the orientation is not known with certainty (**s.a** could be either to the right or to the left of **s.b**, so the choice of the traversing loop would be complicated.

The simplest method for traversing a segment is dividing it into a given number of parts, then proceeding from **a** to **b** by the length of said shorter segments. In the examples, point **p** does exactly that.

There are multiple examples, since the movement from **s.a** to **s.b** can be done in several ways, by using different methods; of course, the solutions will all be similar, given the simplicity of the task.

In the first solution, the trick is placing **p** on **s1.a** first, then projecting it progressively from *itself* towards the second end point **s1.a**. In the image, the points are drawn in **blue**.

```
s1 = Segment(-2,-1,1,2.1)
distance = s1.length() / n
p = Point(s1.a)
for i in range(n+1):
    p.draw(g,BLUE)
    p = p.projection(s1.b,distance)
```

In another solution, each new point is found with the **at** method of segment **s2**, which places **p** by using a proportion on the total length (that's because it matches the segment to the interval **[0,1]**). In the image, the points are drawn in **goldenrod**.

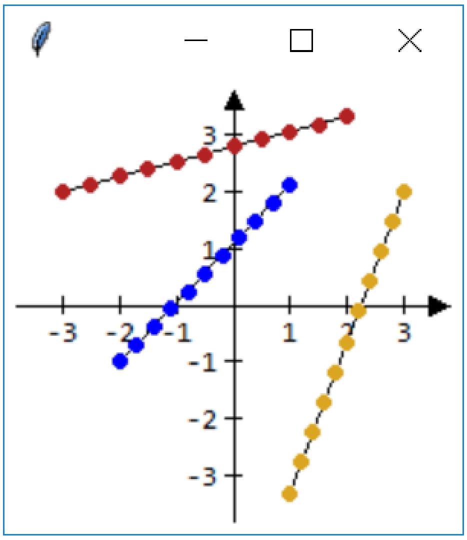
```
s2 = Segment(3,2,1,-3.3)
distance = s2.length() / n
for i in range(n+1):
    x = distance * i / s2.length()
    p = s2.at(x)
    p.draw(g,GOLDENROD)
```

Another solution projects from **s3.a** towards **s3.b** by increasing distances. In the image, the points are drawn in **firebrick**.

```
s3 = Segment(-3,2,2,3.3)
distance = s3.length() / n
for i in range(n+1):
    p = s3.a.projection(s3.b,distance*i)
    p.draw(g,FIREBRICK)
```

Undoubtedly, there may be more methods, but this list is exhaustive enough for our purposes.

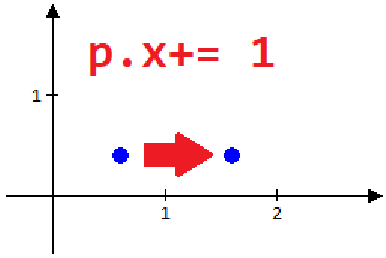
We may notice that, while the number of smaller segments is **n**, the total number of points drawn is **n+1**, not **n**, since the loops cover both end points of the original segments.



USE WORLD, NOT SCREEN, COORDINATES

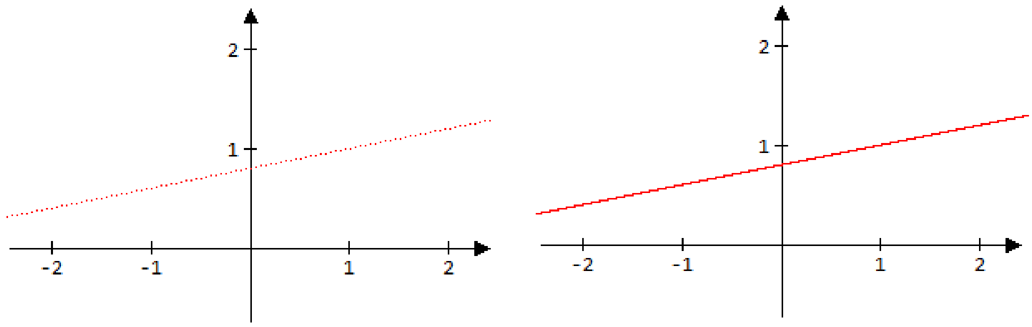
As mentioned before, calculations are done with *world* coordinates. Property **`g.px`** is a *world* distance, *corresponding* to one pixel. **Distances are <u>not</u> measured in pixels**.

At all costs, avoid the temptation of moving a point by **`1`**: if we wrote **`p.x+=1`**, the point would advance by one *world unit*, not one *pixel* (see image).



Approximations

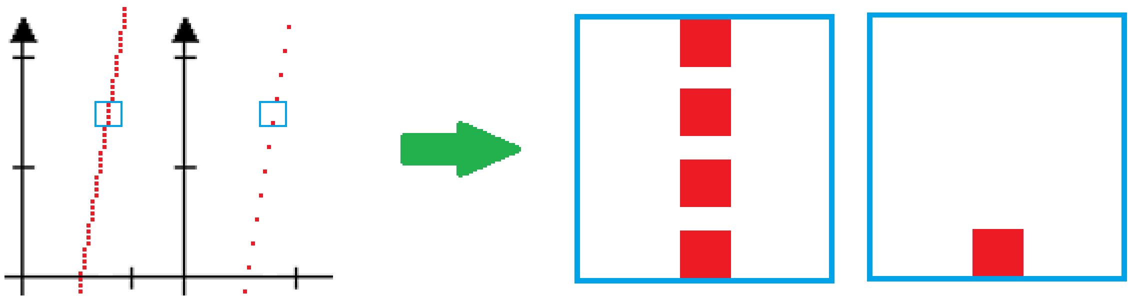
Of course, a graphic window is a square matrix of pixels, so some approximation is necessary: the usual method is to compute the coordinates of a number of points, packed closely enough to give the *appearance* of a continuous line. The example below shows how the distance between consecutive points affects the rendering of the curve:



As in all simulations, the choice of how many points, and how distanced, should follow the terms of the calibration (which is, after all, a world/screen proportion). Normally, points are chosen half a pixel apart, and if the result is not satisfactory, smaller distances are tried, until a better approximation is reached.

A distance "smaller than a pixel" may seem puzzling, but it makes sense for curves with a steep vertical slope. Of course, it is impossible to draw *between* pixels: the trick is that, while some points on the **`x`** axis are approximated to the same pixel, they have different ordinates, so there are *more pixels to be seen*. This is shown in the image below, where a line is drawn with a step of **`1/4`** pixels, as compared with a step of **`1`** pixel: the blue rectangle outlines four points approximated to the same horizontal screen abscissa, but to *different screen ordinates*. At a normal zoom, or far enough from the screen, the curve with more pixels has a much smoother appearance.

Simply put, fractions of a pixel allow to get some extra points.



BEWARE !!! Expressions such as "one pixel apart" should not be taken literally. Once again, calculations must be done in *world*, not *screen*, coordinates. Screen coordinates, besides being integer numbers, are approximated, and should only be used when transferring a shape to the viewport.

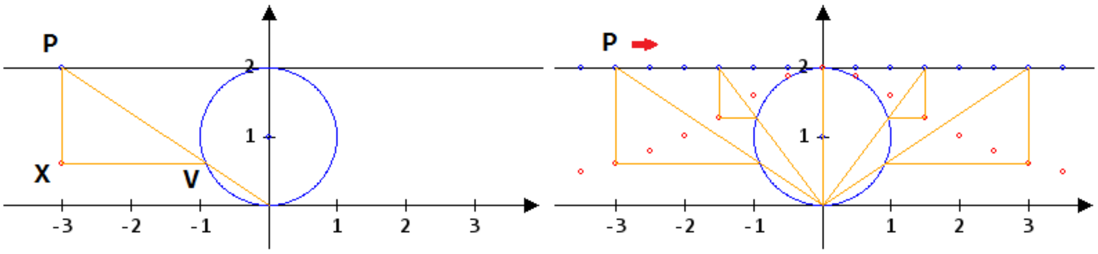
Tracing some historic curves

Example 1: the *witch* of Agnesi

Our first example will be the *witch*. This curve was studied (after Pierre de Fermat in the XVII century) by the Italian scholar Maria Gaetana Agnesi from Milan (1718-1799), one rare instance of a female mathematician in an epoch when women were not even supposed to be anything but wives and mothers (the image to the right shows a plaque fixed to the front wall of the Agnesi mansion near Lecco). In a treatise written in 1784 (*Instituzioni analitiche ad uso della gioventù italiana*), Agnesi used the term suggested by the Italian mathematician Guido Grandi (1671-1742), who called it *versiera*, from the Latin *versoria*, the rope bound to the far end of sails (actually, passing through a metal ring sewn into the sail), used to assist ships during turns. The translator into English, probably John Colson (1680-1760), associated it with the term *adversarius*, as in "God's enemy", traditionally referring to the devil, and, as a synonym, to a witch. Hence the English name.

The simplest construction of the witch starts from a line of equation `y=2` and a circle of radius `2`, passing through the origin, and with the center on the `y` axis; the two curves are tangent in point `A(0,2)`. For any point `P` of the line, its projection through the origin meets the circle at a point `V`. The point `X` with the same abscissa as `P` and the same ordinate as `V` belongs to the witch. This is shown in the first image.

The curve is obtained by *moving* point `P` along the line `y=2` (the *pedal*), then finding tracing the position of point `X`, with geometric (not analytic) methods. The second image shows several of such constructions.



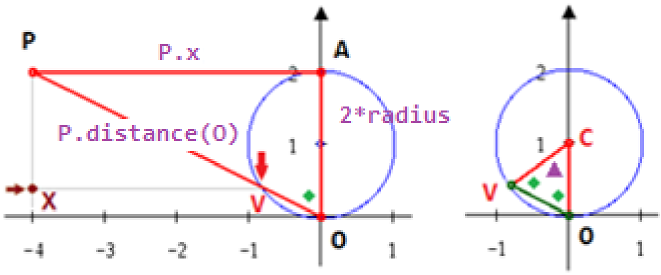
The trick for the geometric solution is to pretend that the points are *already in place*, then finding Euclidean relations between elements of the configuration, so as to locate them.

The first step for finding `V` is solving the square triangle `OPA` (where `O` is the origin, and `A` is the point where the line `y=2` meets the circle), where all the sides are known. The angle marked with a **green** diamond is the key to the solution: it can easily be found with the formula:

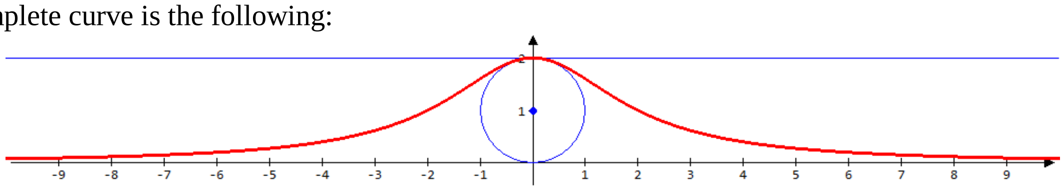
`angle = asin( AP / OP )` (where `AP` is the abscissa of `P`)

Once the angle is known, it is possible to solve the isosceles triangle `OVC` , where `C` is the center of the circle. The central angle (marked by a **purple** triangle) can be obtained by subtraction (`PI-2*angle`); then the length of the chord `OV` can be obtained by the law of cosines.

Now, point `V` *is still unknown*, but it can be obtained by projecting the origin towards `P`, with the length of the chord. Point `X` follows banally, by changing the abscissa of `V` to that of `P`.



The complete curve is the following:

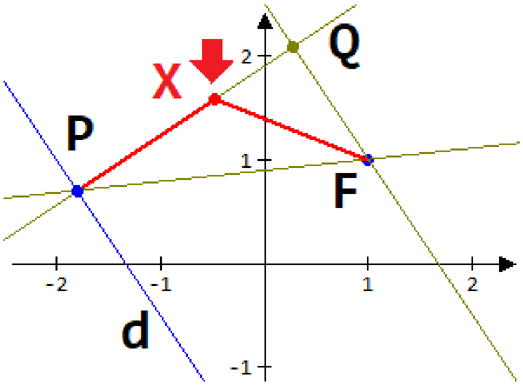


Example 2: the parabola

This is not the classic parabola **y=ax²+bx+c** from analytic geometry, oriented vertically, but an oblique one, which will be built by geometric means, without any algebraic considerations regarding its equation (namely, finding the coefficients **a**, **b**, and **c**).

Geometrically, a parabola is the locus of those points which are equidistant from a point (the *focus*) and a line (the *directrix*). In the image below, point **F** is the focus, and line **d** is the directrix, which is also the pedal line, containing the moving point **P**.

For a point **X** to be on the parabola, it must be equidistant from **d** and **F**. This means that it must lie on the perpendicular to the directrix, passing through **P**, and that triangle **PXF** must be isosceles (see image):



In order to solve said triangle, we can construct a slack triangle, **PQF**, where **Q** is the projection of **P** on the parallel to the directrix, passing through **F**. Knowing all the sides of this triangle (**SSS**), the angle in **P** can be found. Not coincidentally, this same angle is also at the base of the isosceles triangle **PXF**, where side **PX** can be found immediately (being a **ASA** triangle).

Again, the two lines meeting at **Q** (which can be found with the **section** method) are:

```
p.perpendicular(d)
f.parallel(d)
```

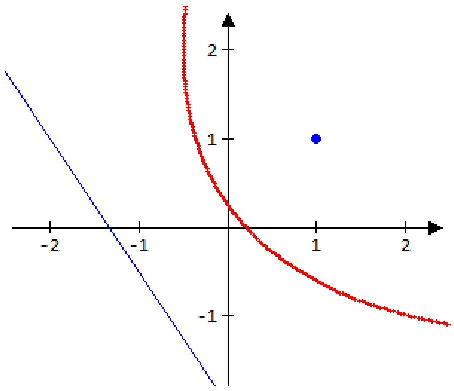
The angles and sides of triangle **PXF** can be found by:

```
x = acos( p.distance(q) / p.distance(f) )      #    Angles at P and F
distance = (p.distance(f)/2) / cos(x)          #    side PX or XF
```

p.distance(f)/2 is a side of the square triangle obtained by projecting **X** on **PF** (which is half of the isosceles triangle **PXF**). Variable **X** is still unknown, but if can be obtained geometrically (with the length of the side just calculated):

```
x = p.projection(q,distance)                   #    construction of X
```

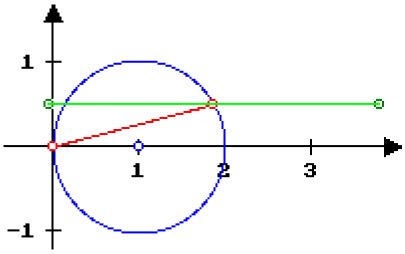
After construction, the parabola is the following:



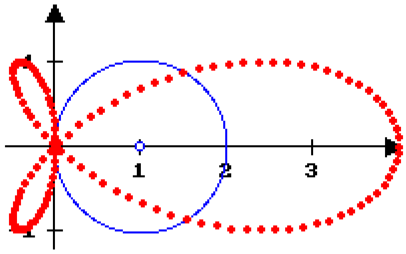
NOTE Another (and certainly simpler) way of finding point **X** is sectioning the axis of segment **PF** with the perpendicular to the directrix passing through **P** (called **PQ** in the first image).

Example 3: the *torpille*

From the storied world of kinematic curves, this one is a beauty. The pedal line is a circle, with one fixed point (the left end of the horizontal diameter: in the example, it is the origin); for each **point** of the circle, two other **points** are determined, moving horizontally in either direction, by the same length as the distance from said point to the origin. In the first image, the three segments have the same length, and the points of the *torpille* are drawn in **green**.



Finding two points of the curve



The final result

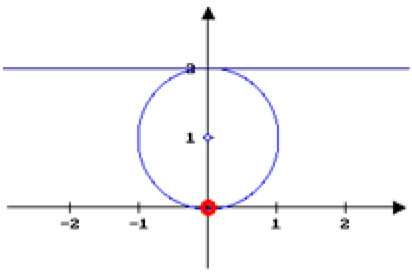
The name *torpille* (a French word) was chosen by the French mathematician Gaston Albert Gohierre de Longchamps (1842–1906), because the curve looks remarkably similar to an actual torpedo (either the fish, the weapon, the fishing implement, or sometimes the *torpedo owl*):



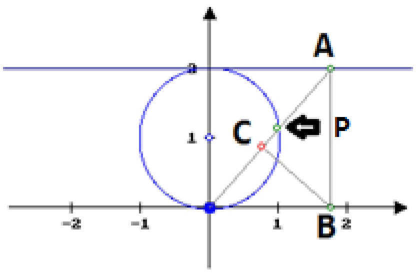

Example 4: the *cissoid* of Diocles

Apparently, this curve takes its name[27] from the Greek word for *ivy* (*kissos*, or *κισσοσ*). It was studied by Diocles (Διοκλῆς), a Green mathematician who lived between 240 and 180 BCE, in relation to the ancient problem of *duplicating the cube*.

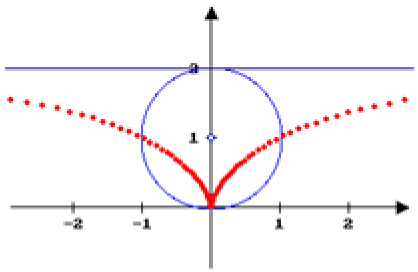
The pedal line is a circle again, with another straight line (tangent to the circle) as a prop, and one fixed point, which is the one opposite the tangent. In this construction, a line is drawn through the fixed point (the origin **O**) and the moving point **P** (indicated by the arrow), determining a point **A** on the straight line. Said point is projected on the x-axis, and the result **B** is projected towards the **OP** line, crossing it in **C**. **C** is the generic point of the cissoid. Incidentally, **OP** has the same length as **AC**, and **OC** the same as **AP**.



The playing field



Finding one point



The final result

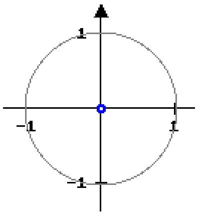
[27] Many English speakers pronounce the word with a soft "c", belying its Greek origin.

Example 5: the *trisectrix* of Ceva

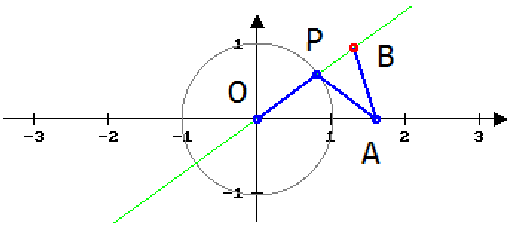
This curve was studied by the Italian mathematician Giovanni Ceva (1647-1734), who moonlighted as a hydraulic engineer and had some clever insights on calculus. Ceva taught and died in Mantova.

This curve is called *trisectrix* because it can be used for the solution of the historic problem of trisecting the angle. The pedal line is a circle again, with another straight line (passing through the center) as a prop In this construction, the prop is the x-axis; a line is drawn through the fixed point (the origin O) and the moving point P (indicated by the arrow). Points A and B must lie, respectively, on the prop and on line OP, in such a way that the three segments OP, PA, and AB are all congruent to one another.

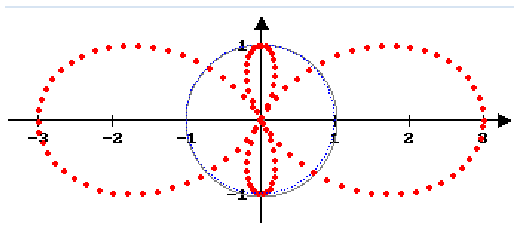
While point A is easily found, solving the isosceles triangle PAB may seem a bit tricky, but, by the exterior angle theorem, BPA is easily found as twice the value of angle POA (the triangle POA being isosceles). PAB is also isosceles, on base PB: so each half of PB can be calculated as PA*cos(2α), and PB is twice that distance. PB can be used to project P towards the exterior of the circle to obtain the position of B.



The playing field



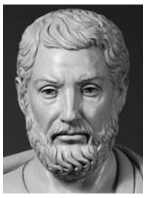
Finding one point



The final result

Example 6: the *quadratrix* of Dinostratos

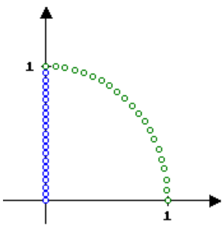
This curve was studied by the Greek mathematicians Hippias (who lived in the 5th century BCE) and Dinostratos (Δεινόστρατος, approx. 390-320 BCE). A similar one was studied by the German mathematician Ehrenfried Walther von Tschirnhausen (1651-1708).

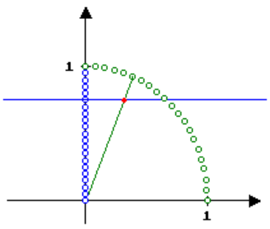
The curve was called *quadratrix* because it can be used for the solution of the historic problem of squaring the circle (of course, not with straightedge and compass). In the Greek construction there are two pedals: a quarter of a circle c, and the corresponding vertical radius. The two points start from the same position (0,1 for simplicity), and they must reach the x-axis simultaneously (of course, the point on the arc travels faster, having a longer distance to cover).

For each couple of points, the **blue** one A is on the segment, and the **green** one B is on the arc. The corresponding point on the *quadratrix* is obtained by sectioning the horizontal line passing through A with the one joining B and the origin. The coordinates of the point on the x-axis are (2*c.r/pi,0).

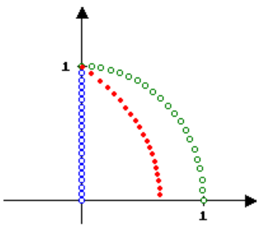
Von Tschirnhausen's construction is a little bit different, using a vertical line passing through B.



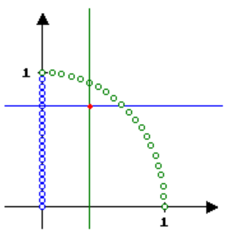
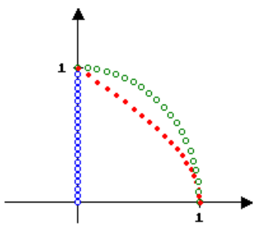
The playing field



Finding one point



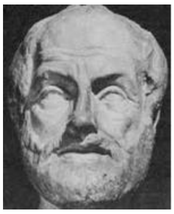
The final result

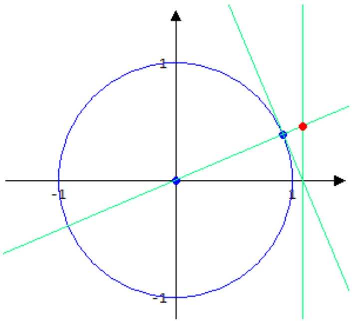
Von Tschirnhausen's construction

Example 7: the *kampile* of Eudoxus

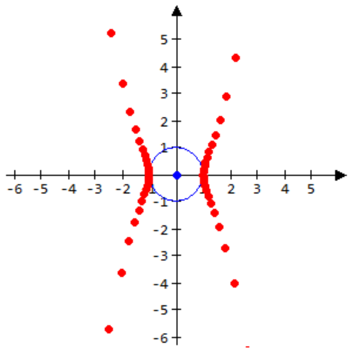
This curve was studied by the Greek mathematicians Eudoxus of Cnidus (Εὐδοξος, 408-355 BCE), in relation to the ancient problem of *duplicating the cube* (again, not with straightedge and compass).



The pedal is a circle. For each point of the circle (drawn in **blue**), find the section of the corresponding tangent with the line determined by the horizontal diameter (which can be the **x** axis). Then draw the perpendicular to said **x** axis through that point, and find the intersection with the prolongation of the radius (these two lines are shown in **green**).



Finding one point



The final result

Example 8: spirals

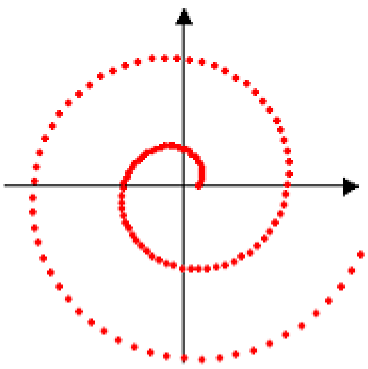
A spiral is described by the far end of a rotating segment, while the segment itself is stretched as it rotates. There are several spirals, depending on how the segment is stretched. The classic ones are:

- The *Archimedean spiral*, where the stretching is uniform (at each step, the length of the segment increases by the same amount as the angle of rotation;
- *Galileo's spiral*, where the motion of the point is accelerated;

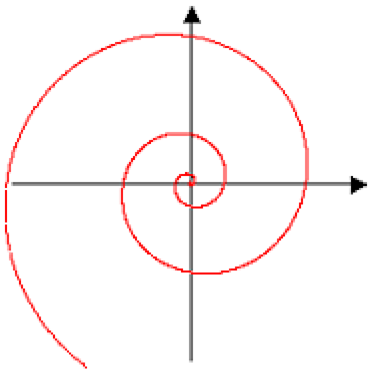
There are other spirals, but the basics remain the same. Among them:

- Fermat's spiral
- Euler's (or Cornu's) spiral
- The Lituus
- The logarithmic spiral

The Archimedean spiral shows the beauty of the radian angle, which is the measure of the arc relative to the radius: so, in a circle with radius one, the angle and the arc coincide.



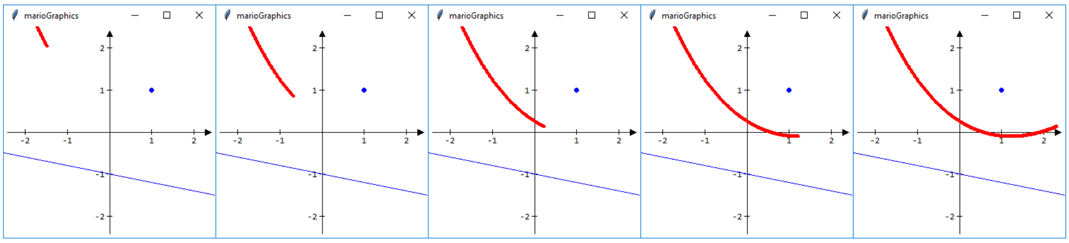
Archimedean spiral



Galileian spiral (strongly enlarged: **`2000x2000`**)

Example 9: two studies in animation

The construction of curves can be made more appealing by watching the curves as they are drawn by the designer. Let us consider the tracing of a parabola from example 2. Some moments of the construction are shown in the following images:

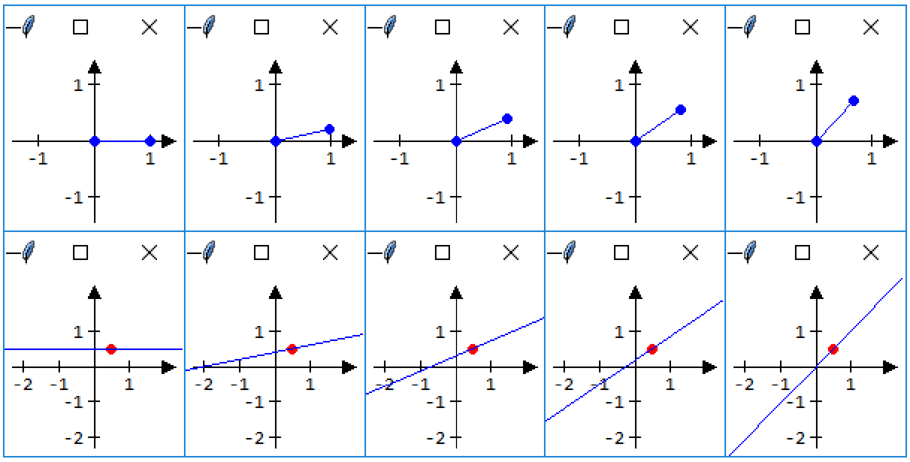


For this, the designer offers a method called **tick(n)**, which calls the system timer with a given interval:

```
p.x=g.l
while p.x<=g.r:
    p.y = d.ordinate(p.x)
    t1 = p.perpendicular(d)
    s = Segment(p,f)
    m = s.midpoint()
    t2 = m.perpendicular(s)
    x = t1.section(t2)
    x.draw(g,RED,2)
    p.x+=g.px
    g.tick(.1)
```

By properly adjusting the time interval **n**, we can watch the construction at any speed.

This simple animation, however, only consists in adding shapes to the viewport. Other animations are slightly more complex: for example, with rotating lines or segments, the shapes change and must be *redrawn*, as in the following sequences:



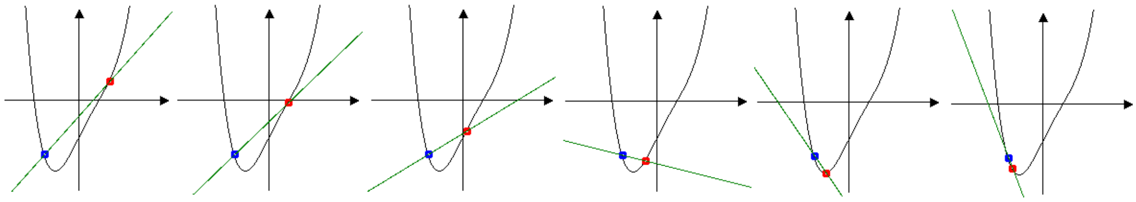
A segment can just be spun on its end point **a** (or **b**, if needed) by a given angle. A line, being infinite, must be recalculated and redrawn. The two designer methods are called, understandably, **spin** and **redraw**.

The two following snippets should give at least an idea:

<table>
<tr><th>Segment</th><th>Line</th></tr>
<tr><td><pre>
s = Segment(0,0,1,0)
s.draw(g,BLUE)
for i in range(20):
    s.spin(.2)
    g.tick(.1)
</pre></td><td><pre>
for i in range(20):
    s.rotate(.2)
    z = Line(s)
    t.a = z.a
    t.b = z.b
    t.c = z.c
    t.redraw()
    g.tick(.1)
</pre></td></tr>
</table>

Example 10: the derivative

Rather than a curve, this animation, involving a function, is the generation of the tangent in a given point of the curve (a process used in evaluating the first derivative of a function).



The process is quite simple,. We need one fixed point (where the tangent is to be found), and one moving along the curve towards the former: at each step, the secant line joining the two points gets closer to the tangent (at each step. calculating a new line, then redrawing it as we have just seen). From calculus, it is known that the slope of the tangent line is also the value of the derivative of the function in the given point.

Finding the derivative of a polynomial with calculus

With polynomials, it is quite simple to find the derivative algebraically[28]. The only consideration necessary is that each coefficient of the polynomial is an element of an array, with the position corresponding to the degree of the independent variable. When a **`Polynomial`** object is declared, it acquires a list[29] property containing the coefficients, called **`x`**, with as many elements as the degree of the polynomial, each corresponding to the power of the independent variable. With the preceding example of an object **`p`** declared as $y=x^4-x^3+2x-1$, said list becomes:

```
p.x[0]=-1; p.x[1]=+2; p.x[2]=0; p.x[3]=-1; p.x[4]=+1
```

Remember: the list only contains the ***values*** of the coefficients (those shown in ***red***). Do not fall in the trap of considering the **`x`** as part of the structure.

A new polynomial containing the first derivative of **`p`** can be built from said coefficients. This new function can be evaluated in any given point, and the result can be compared with the one obtained with the numerical method. Recalling the formula for the derivative of a generic polynomial term:

Formula for the derivative of a generic polynomial term	$\frac{d}{dx}\mathbf{x^n} = \mathbf{n} * \mathbf{x^{n-1}}$
The inverse formula, in array notation	`q.x[i]=p.x[i+1]*(i+1)`

From calculus, we know that the derivative polynomial is one degree lower than the given one, with:

```
q.x[0]=1*p.x[1]; q.x[1]=2*p.x[2]; q.x[2]=3*p.x[3]; q.x[3]=4*p.x[4]
```

Instead of writing a program, we can use the **`derivative()`** method associated to **`Polynomial`** objects. And there's more: with a **`Rational`** object, we basically have two **`Polynomial`** grouped together, one for the numerator and one for the denominator, so we can end up writing something like the following:

```
p = Polynomial(-1,2,0,-1,1)     #   y = x⁴ - x³ + 2x - 1
p.derivative().show()
r = Rational((3,2,1),(2,5))     #   y = ( x³ - 2x² + 3 ) / ( 5x + 2 )
r.show()
r.numerator.show()              #   This is a "Polynomial" object
r.denominator.show()            #   This is a "Polynomial" object
r.numerator.derivative().show()
r.derivative().show()

4 x ** 3 - 3 x ** 2 + 2         #   THE RESULTS ON THE SHELL WINDOW
x ** 2 + 2 x + 3 / 5 x + 2      #   Rational r
x ** 2 + 2 x + 3                #   Numerator
5 x + 2                         #   Denominator
2 x + 2                         #   Derivative of the numerator
5.0 x ** 2 + 4.0 x - 11.0 / 25.0 x ** 2 + 20.0 x + 4.0  # Derivative of r
```

---

[28] With rational functions, the procedure is more complex; with general functions, it is generally impossible.

[29] Often called an *array*.

GIS

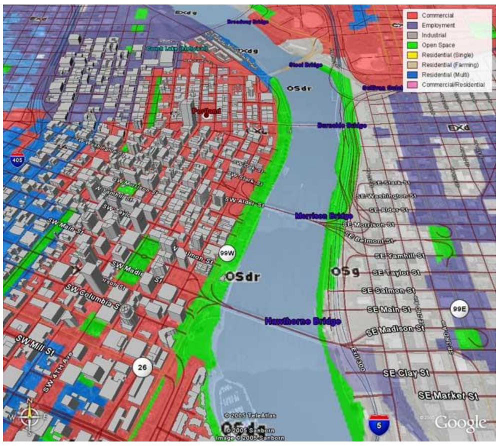
or:

an application of geometry

Geometry Systems

What, if anything, is GIS?

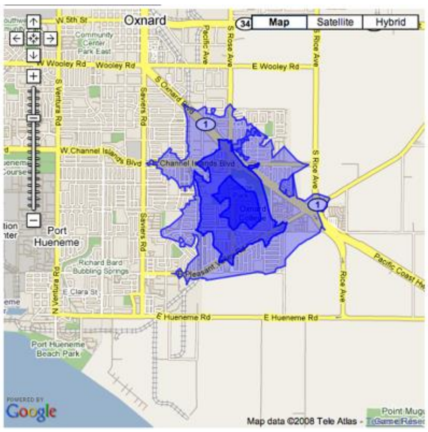
Drawings, and more



Drawings have been around for a long time in computer science. Graphic problems in engineering and surveying, which share a common base in geometry, led to the development of **CAD** (**C**omputer **A**ided **D**esign) technology since the 1980s. Nowadays, **CAD** users have scores of powerful tools and applications that help them produce accurate and meaningful drawings. Graphic interfaces are used to interactively trace lines and curves; geometric data may be read from text files, and then converted into graphical entities; data may also be obtained by analyzing scanned images, or digital pictures taken from aircraft or satellite. State-of-the-art, high-resolution peripherals give excellent visual feedback and printouts.

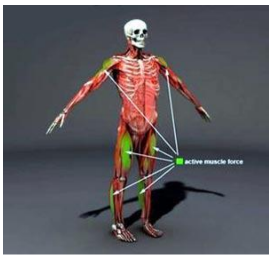
CAD and GIS

Along with technical improvements in hardware and software, **CAD** applications were soon extended beyond the actions of drawing and map printing, leading to the integration of graphical and textual information: this resulted in the creation of hybrid systems, containing both kinds of data, which is commonly referred to as a *graphic database*. This solution is widely used by municipal or government agencies dealing with land surveying, environmental resource management, property, town planning, and any other subject related to maps. These users most often visualize, print and analyze graphic files made by external consultants, and need instruments to automate such procedures, or to customize their analysis on both graphic and text data. Also, this information about public services could be made available through the Internet, thus reducing the need of direct communication with the offices and cutting down on response times for most queries: but while displaying maps on Web sites is no big deal, linking them in real time to related data can be quite a job.

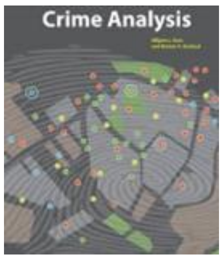


The solutions to these problems require that the graphical nature of **CAD** be applied to geography and to the management of related data, making all components available for analysis and inquiry, both for the local and the remote user: a computer system for storing, manipulating and analyzing maps and geographic data is called **GIS** (**G**eographical **I**nformation **S**ystem). In a **GIS**, geometric entities and properties represent anything that can be spatially referenced: thus, lines, polygons, lengths and areas may represent roads, land parcels, mileage, and, most important, taxable land.

Over the years, the acronym **GIS** has been subject to several expansions, though with the same general meaning: "**Geographic Information System**", "**Geographical Information System**", "**Geospatial Information System**"**.** The same holds in Italian, where the acronym **SIT** has been expanded as "Sistemi di Informazione Territoriali" or "Servizi di Informazione Territoriale", with some minor gender and number changes.



While the most common integration of graphics and data occurs in geography, this is not the only field of application. GIS software is flexible enough to treat any kind of data, provided it is spatially referenced: heating and sewage systems, crimes and suspects, even the human body, have all been mapped and analyzed. So the same considerations done on **GIS** may be applied, with due changes, to all graphic databases.



How it all got started

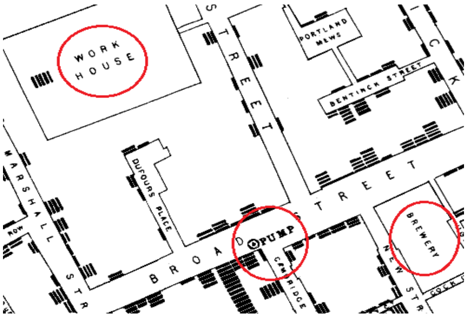
Some credit the English physician John Snow (1813-1858) with the first georeferenced study. In late summer 1854, Snow, who practiced in the Soho district of London, had to face a cholera outbreak in that very zone.

Snow

Adopting an unusual procedure, he took a map of the surroundings, then used points to represent the locations of the recorded cases. Working with the local parson, Henry Whitehead, who knew most of the residents, he could trace the source of the disease: a water pump which had been contaminated with water leaking from the house of the first victims (a policeman and his family). He then asked the local authorities to seal the pump by removing its handle: though reluctant to believe him, they met the request out of desperation. Following this action, the cholera outbreak, which had already begun to dwindle, stopped outright.

The John Snow map

By observing the points on the map, Snow had noticed that most of them clustered near a water pump in Broad Street (now Broadwick): he then found out that many people in Soho took water from there, as it was considered the best in the neighborhood. Some people who died outside London were traced to having visited Soho days before falling sick. On the other hand, only a handful of the 535 inmates of the Poland Street workhouse[30], which was just around the corner from the pump, contracted cholera: Snow found out that the facility had its own well. Furthermore, none of the 70 workers of the Broad Street brewery got sick: being given an allowance of beer every day, they hardly drank any water[31].



Detail with locations of workhouse, pump, and brewery (left to right).

These consideration, sparked by the study of geometric patterns in the distribution of cholera cases, led Snow to his groundbreaking finding that the disease had its source in contaminated water: he could not identify the agent, but he correctly argued that the solution lied in better sanitation.

Pacini

Curiously, the bacterium responsible for the epidemic (*Vibrio cholerae*) was discovered in that same year 1854, by the Italian anatomist Filippo Pacini (1812-1883), but the link between the two would be discovered only years later by Robert Koch (1843-1910). For his extensive work on epidemics, Koch was awarded the 1905 Nobel Prize for Medicine and Physiology.

Koch

Until then, the still prevalent “miasmatic theory” attributed contagion to the inhalation of “unhealthy” air. This was reflected in the word *malaria*, which means “bad air” in Italian, while the actual cause of that disease is a bacterium called *plasmodium falciparum*, which is carried by the anopheles mosquito. The finding that cholera was water-borne was one of the steps towards the recognition of parasites[32] as the culprits for infectious diseases.

To this day, a pump without a handle stands on the place of the original pump, as a monument to John Snow, who is also honored by a nearby pub (The John Snow) bearing his name. The position of the pump used to be marked by a pink granite kerbstone, which is now part of the sidewalk around the memorial.

Vibrio cholerae

Plasmodium falciparum



The pump, pub, and kerbstone in Broadwick Street

[30] Workhouses were institutions where the destitute could receive food and lodging in exchange for work. They hardly were humanitarian places: life was supposed to be harsh, so it was like a sort of debtors’ prison.

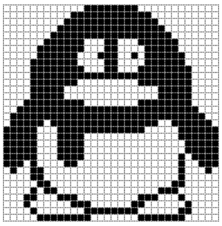
[31] Throughout history, drinking water has been contaminated with all sorts of infectious micro-organisms: until proper sanitation was introduced, fermented beverages were much safer, at least from this point of view.

[32] These include viruses, bacteria, fungi, and worms.

Handling drawings

Raster pictures

The simplest images that can be displayed on a computer screen are *raster* pictures. A raster picture is actually a matrix of colored dots (pixels), which can be produced with many programs[33], and stored in several formats[34]. The appearance and quality of the images we see in a raster picture are determined by its *resolution*: with a low resolution (or by zooming in a picture), it is possible to see the pixels as squares. A raster picture is analogous to a video display, both being composed by a grid of pixels (the very term "raster" derives from the Latin word "rastrum", meaning "rake", and has also been used to describe the scanning process used in **CRT** technology). Its size is independent from external factors, such as screen hardware and video drivers.



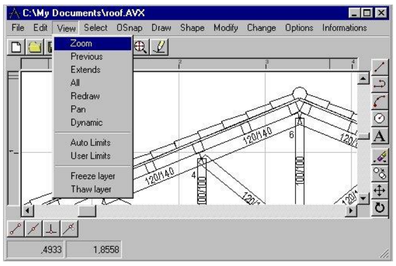
Mario the penguin

Background drawings

The management of graphic objects is done using a raster picture, which provides a physical base of pixels: each graphic object will be drawn on its corresponding group of pixels on the raster, which for this reason is called a *background*. The background can be an actual picture, taken from an image file, or a virtual one, composed only by white pixels, in which case it will be called a *blank* raster. Any image or drawing can be displayed as background, either on an existing raster, or automatically generating its own blank raster.

Vectorial drawings

In a raster picture, patterns can be detected visually, but cannot be managed as geometric entities: pixels are drawn independently from one another, and are not grouped into graphic objects, such as segments, circles, and the like. These entities, which form drawings usually handled with **CAD**[35] programs, are also called "vectors" (hence the expression "vectorial drawings"). While raster pictures are fixed collections of pixels, vectors are characterized by their *geometric* properties: thus, a segment is identified by the coordinates of two points, and only when drawn it will be converted into pixels. Drawings come in a lot of formats; in the engineering fields, the most used are **DXF** (**D**igital e**X**change **F**ormat), **SHP** (ESRI **SH**a**P**eFile), and **DGW** (AutoCAD **dr**a**w**in**g**).



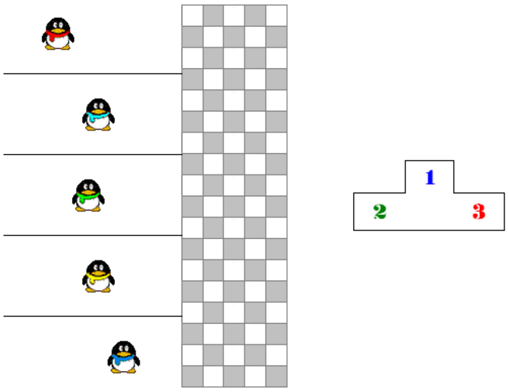
Simulations

In preceding chapters, we have dealt with the problem of calibration, defined as the correspondence between *world* and *screen* coordinates. The term *world* has not come into use by chance: a graphic window may represent an actual event, taken from the real world or from imaginary ones.

So, it is possible to write a program tracing the movement of a projectile through the air, or depicting the daily contagions of the 1854 cholera outbreak; but also, with a bit of imagination, penguins racing, or pigs flying.

After calibration has correctly placed the components onto a graphic window, what is necessary is a set of *game rules*, that is, an algorithm that determines the flow of the event, so that it can be followed on the screen.

All this is called *simulation*, the topic of the next chapter.



Penguins do not usually wear scarves, nor do they race towards a checkered finish line with a podium awaiting them, but, with proper calibration and animation, that could look at least *plausible*.

---

[33] There are scores of such programs, called *image editors*. A very popular one in the free software community is **GIMP** (**G**nu **I**mage **M**anipulation **P**rogram).

[34] The most common formats are: **BMP** (**Bi**t**m**a**p**), **GIF** (**G**raphic **I**nterchange **F**ormat), **JPG** (actually **JPEG**, **J**oint **P**hotographic **E**xperts **G**roup), **TIFF** (**T**agged **I**mage **F**ile **F**ormat), **PNG** (**P**ortable **N**etwork **G**raphics). Another format, called **MrSID** (**M**ulti**r**esolution **S**eamless **I**mage **D**atabase) was developed expressly for **GIS**.

[35] **CAD** stands for **C**omputer **A**ided **D**esign.

SYSTEMS AND SIMULATIONS

List of simulations

GIS Cellular automata

What, if anything, is a system?

Scientists like to organize the objects of their studies, classifying them as *systems*. The word has no conclusive definition, but one commonly accepted description of system is “a set of interacting entities, connected by reciprocal relations”, which specifies that:

- A system has components;
- the components work together (either for a purpose or just as it happens).

Some summarize these concepts with the formula:

System = Attributes + Relations

where *attribute* is another term for *component*, and *relations* pertain to how components *interact*.

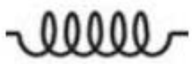
How a system is defined depends on how it is studied. Consider a refrigerator, an artificial system: the casual user could list its attributes as “box, door, handle, plug”; the manufactures has other concerns, having to put together “compressor, condenser, expansion valve, evaporator”; the designer worries about “materials, shapes, colors”. Just the same, differences can be analyzed for every system, which are therefore defined by how they are dealt with, or how they are observed.

One of the most important operations in the study of systems is *modeling*. Models are variously used to describe and clarify systems, or to predict their behavior: they can be used in teaching, in instruction manuals, or as guidelines during construction (with the objective of saving time and money).

Models are often classified according to their practical use:

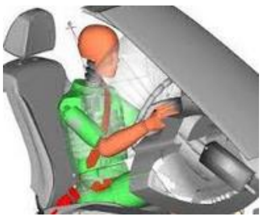
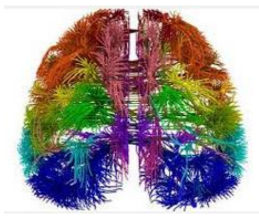
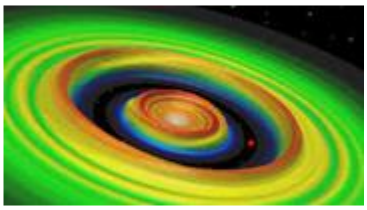
- Iconic – scaled-down representations (as with toy cars or trains), or scaled-up ones (as with colored marbles strung together to show molecular structures);
- Graphical – sketches, drawings, or projects;
- Mathematical – sets of formulas or equations describing the behavior of a system (as Newton’s laws do).

All these are descriptions of systems, sometimes made with *analogies*, as often happens in physics: for example, springs in mechanics and inductances in electricity are depicted with the same “coil” icon, because of their similar behavior.



It is very frequent to put a mathematical model to work on a computer: this is what they call a *simulation*, where results can be shown as data, or, better yet, as *videos* that represent the dynamic processes of the system in an *animated* sequence.

Models and simulations are frequently used during instruction or design, where operating (or destroying) actual systems would be too expensive to afford, or unethical, or downright impossible:

Sometimes it could be just for fun:

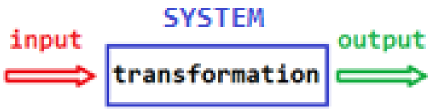
Videogames are often considered the toughest software to write, and not without reason. Consider an inventory system: after describing items, storage, and transactions, and preparing a few printed summaries, the job is more or less done. Videogames, on the other hand, must describe strange worlds with consistency, and, what’s more, do so in an entertaining way, or users would soon turn to other products.

Classification of systems

Systems can be classified according to structure:

1	NATURAL	Already present in nature: tides, winds, planetary systems. These systems have no design (unless someone believes in a creator).
	ARTIFICIAL	Conceived or built by man for public or private purposes: the school system, justice, railway networks, families, sewers.
	MIXED	When there are interventions on natural phenomena: seawater desalination systems, dykes, water mills. A beaver dam qualifies as mixed (or not?).
2	CONCRETE	Physically existing: natural systems, computers, oil tankers, refrigerators. They can be touched and measured.
	ABSTRACT	Conceptual systems, devised by man: calendars, algebras, governments. Most of these creations measure concrete systems.
3	FINITE	All attributes are quantifiable with a discrete number of values.
	INFINITE	At least one attribute has an infinite range (it should be noted that an infinite system, in order to be studied, must be *discretized*, that is, approximated with a finite system).

Conceptually, the behavior of a system is visualized with a *black box*, taking *input* signals and, as a result, generating *output* signals, in a process called *transformation*. The term *black* specifies that the details of the inner workings of the systems are hidden from the observer: it does not refer to the diagram, where the box is usually white.



Systems are classified according to their behavior in a number of ways, some of which are listed below:

1	OPEN	exchange information with the environment: thermic machines, living systems, computers (*information* is an umbrella term for any data involved)
	CLOSED	do not interact with the outside: a bank account, geometry
2	STATIC	do not evolve, as, for example, grammars and algebras; rules may be added or modified, but this happens over long periods of time
	DYNAMIC	evolve in time, un an observable way, as most concrete systems do
3	DETERMINISTIC	Output is a (more or less) predictable function of input (computers, billiards, guns)
	STOCHASTIC	Output is not predictable (coin tosses, dice throws, roulettes, card games)
4	LINEAR	React proportionally to input variations (scales, speakers). Physical systems normally have a range of acceptable inputs, outside which they malfunction or break down.
	CHAOTIC	Sensible to **boundary conditions**, infinitesimal input variations have macroscopic, unforeseeable reactions, as in a cloud of smoke, or in a turbulent motion. A beehive is normally described as chaotic, subject to Brownian motion (at least for us).
5	MEMORYLESS	Same response from same input (no memory), as in many electronic circuits
	WITH MEMORY	Output depends on both input and current state. An elevator moves according both to the requested floor and on which floor the cab is in a given moment.

Of course, computer simulations are used in the study of *dynamic* systems, so that their evolution can be analyzed and/or visualized by some software. In the study of systems there is a considerable amount of theory, with scores of rules and definitions, clarified by with diagrams and tables that describe how systems work. Most of it is not essential for this course: it suffices to consider that, in any given moment, a system is in a certain *state*, which is measurable by the values of its attributes. In a computer simulation, which is, after all, a program, attributes correspond to *variables*, whose values describe the state of the system. Not surprisingly, they are called *state variables*.

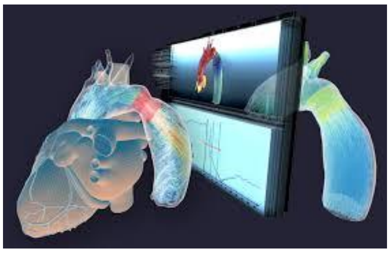
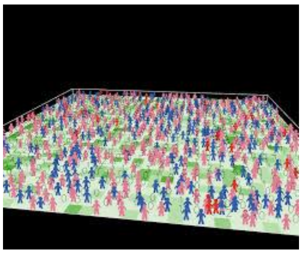
Dynamic systems, in their evolution, can reach a particularly significant situation, called *homeostasis*, which is, essentially, a condition of equilibrium that can be sustained during time. In the case of natural systems, many have exhausted their homeostasis, for one reason or the other, and we can only observe their fossil remnants: after all, most paleontologists observe that about 99% of all species have gone extinct. As for artificial systems, many a frustrated user is well aware that they are subject to the opposite of homeostasis, that is, failure and malfunction. This, at least for some observers, may be part of a manufacturing strategy called *planned obsolescence*, that is, designing products with a limited life, having them, before long, either become unfashionable or just break down (preferably, right after the warranty period has expired). Whatever the reason, failure is when the value of a project can be tested: any idiot can anticipate what happens when a device works properly, but only the prepared mind reacts in times of trouble.

A simple world of simulations

Computer simulations can be quite sophisticated, as moviegoers can attest, when watching superheroes zipping through the air or spectacular crashes taking place:


All it takes is breaking down the objects or the characters into a huge number of elementary, interconnected geometric shapes (the most useful being *triangles*), then moving them according to the desired effect (which may, or may not, follow the laws of physics):




Of course, we cannot dream of achieving such heights, this being an introductory course. So our worlds will be populated by simple imaginary creatures, moving in rather featureless backgrounds.

On the sea, we will follow the adventures of Mario the penguin (maybe with Maria the lady penguin), Sushi the fish, and Clyde the chicken (on a raft):


Mario the penguin


Maria the lady penguin


Sushi the fish


Clyde the chicken

Whizzing or gliding through the air, we will see Willy the pig, Melanie the bee, and Eric the half-a-bee[36]:


Willy the pig


Melanie the bee


Eric the half-a- bee

With these and other actors, we will explore the basics of computer animation.

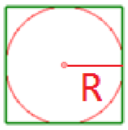
[36] If you do not know about Eric, then you should do a little research on Monty Python. Incidentally, Guido van Rossum borrowed the name of the Python language from that group.

Computer simulations of simple systems

Example 1: approximating the value of π (back)

π used to be a geometer's nightmare, but is now one of the props of the mathematical sciences. In this case, it will only be an excuse to introduce the so-called "Monte Carlo methods", a class of computational algorithms that rely on random numbers to simulate a probabilistic event. The methods and the peculiar name are credited to Stanislaw Ulam (1909-1984) and Nicholas Metropolis (1915-1999), scientists who, among other things, worked at the Manhattan Project in the 1940s. The name derives from casinos, where chance and randomness are dominant (Monte Carlo, apparently, was where Ulam's uncle gambled).

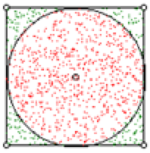
Many methods for evaluating π have been studied. Archimedes, for one, used a geometrical method involving the perimeters of polygons inscribed and circumscribed about a circle. In the following centuries, scores of algebraic methods, more or less efficient, have been devised. This Monte Carlo simulation, which belongs to the computer age, is a *statistical* method involving a square circumscribed around a circle with radius **R** (or any radius, for that matter, since **R** gets kicked out of the equation).



The method uses a pair of algebraic formulas obtained from plane geometry:

Area of circle	$A_C = \pi * R^2$	
Area of square	$A_S = 4 * R^2$	both formulas contain R^2, so $A_C / \pi = A_S / 4$
Final proportion	$\pi / 4 = A_C / A_S$	note that the radius has quietly disappeared
Final formula	$\pi = 4 * A_C / A_S$	

Since the area of the square is known, and **4** is a constant, the problem is reduced to *estimating* the area of the circle (of course, referring to π is out of the question, since it is the value we are trying to compute).



Imagine the two shapes on a dartboard, centered on the origin: for simplicity, since the radius does not count, we can set **R** to **1**, so that the shapes range from **(-1,-1)** to **(1,1)**. Then, imagine throwing a number of darts, all landing in the square: some will land inside the circle as well. The fraction of "inside" darts on the total number of darts thrown is the same proportion as the area of the circle, relative to the area of the square (hence, the value of $\pi/4$).

The darts can be simulated with points, chosen at random, with coordinates ranging from **-1** to **+1**. The landing of a dart in the circle can be verified with the **inside()** method (called either to the point or to the circle), and marked with a **red** pixel (the other darts will be drawn in **green**). Variable **p** should be declared as **pixel**, so as to be shown with a single dot, instead of a small circle. The Python function **uniform()** generates a **double** random number in the half-closed[37] interval **[-1..1)**, or **(-1..1(** for some.

In the following snippet, we will use a simple point generation loop, with no user interaction.

```
r = Rectangle(-1,1,2,2)
c = Circle()
p = Pixel()
darts = 1000
inside = 0
for i in range(darts):
    p.x = uniform(-1,1)
    p.y = uniform(-1,1)
    if p.inside(c):
        g.draw(p,RED)
        inside+=1
    else:
        g.draw(p,GREEN)
    g.tick(.01)
```

> **NOTES**
>
> **p.inside(c)** and **c.inside(p)** are dual, yielding the same result.
>
> We may want to add a delay (**tick**), but with a large number of point this could be unnecessary.
>
> We will observe that the more points are generated, the better the approximation will be: after all, with just one point the estimate for π would be either **0** or **4**.
>
> Of course, Python already has a magic constant called **pi**, which we can compare with our estimate.

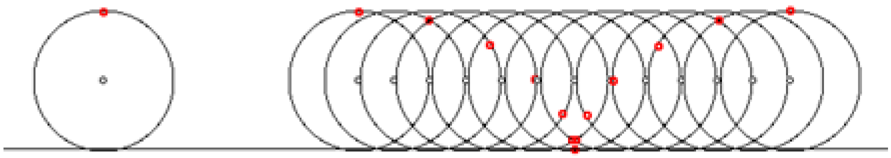
```
PI = 4.0 * inside / darts           #   The final result of the proportion
write("Result:", PI)                #   Show it on the shell window
write("magic constant:", pi)        #   Show the Python value
```

---

37 The interval is open to the right (which means that the end point is not included, or **0<=x<1**).

Example 2: the wheel **(back)**

While in the preceding example the main shapes stood still[38] during the proceedings, here a true animation will be shown: a circle, representing a wheel, will travel along a line, completing one full rotation.

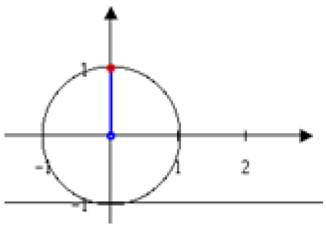


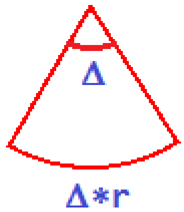
As is well known from physics and geometry, any fixed point on the circle draws a curve called *cycloid*:

The elements for this simulation are:

- one circle (the wheel);
- one horizontal line, tangent to the circle (the ground);
- one segment, describing one radius of the circle (in the wheel, a spoke);
- one point on the rim of the circle (the far end of the spoke, or the valve of the bladder).

The spoke and the valve are used to show the rotation of the wheel, since the circles are indistinguishable from one another. The spoke segment may be drawn or not, depending on preferences. In this example, a default circle has been chosen (centered in the origin, with radius **1**), and the ground line is **y=-1** (its coefficients are **0,1,1**); yet, any other positioning would be just as good, since the objects will be moved *geometrically*, not algebraically.





From geometry, we know that the length of an arc with radius **r** and central angle **Δ** (delta) is equal to **Δ*r**, when measured in *radians* (which is the case with most graphic libraries). If the radius is **1**, the length of the arc is **Δ**, which corresponds to the distance by which the entire circle **c** must be moved, thus assimilating the rotation to a linear motion. Segment **s** also rotates by the same angle, and the "valve" point **v** follows the far end of the segment (which is point **s.b**, since **s.a** follows the center of the circle). Once again, we will briefly pause the program at each step (more on this later).

The distinctive feature of this animation are:

- the circle (wheel) and the segment (spoke) are moved by the same distance;
- the spoke is spun by the same angle associated to the rotation of the wheel;
- the point (valve) is moved to the final position of the far end of the segment;
- the point is redrawn, because when the segment is spun, it ends up covering the point, due to how the designer handles spinning segments.

```
delta = pi / 100              #   rotation angle
a = 0                         #   rotation memory
while a<=2*pi                 #   one full rotation
    g.pause(0.01)             #   pause animation (same as "tick")
    c.move(delta*c.r,0)       #   move wheel
    s.move(delta*c.r,0)       #   move spoke
    s.spin(-delta)            #   rotate spoke (clockwise)
    v.go(s.b)                 #   move valve to new position
    v.redraw()                #   draw valve again
    a+= delta                 #   total angle
```

[38] *The day the Earth stood still* is a 1951 movie, based on the 1940 short story *Farewell to the master* by Hiram Bates.

The basics of animation (back)

What we have just seen is nothing new: it can be considered an extension of the study of geometric curves, with the addition of a bit of imagination. Clearly, we need something more powerful and flexible, adapting our programs to the study of general systems, including those that are not easily predictable. Real-life situations are normally much more complex than one single wheel: even the simplest piece of machinery has many interacting parts, and when we arrive at studying the behavior of such things as aircraft and weather systems, the number of components is astronomical.

Piero Caldirola

Following theory, the evolution of a dynamic system is normally studied by evaluating the state of the system at given time intervals, called *chronons* in the trade. The term *chronon* comes from quantum mechanics, where it was introduced in 1927 by the Frenchman Robert Levi (of whom little is known), then redefined around 1980 by the Italian Piero Caldirola (1914-1984), who expressed it as 6.97×10^{-24} sec. Of course, our intervals will be much longer, the computer clock (and **`Tkinter`**) being unable to reach below a hundredth of a second – just consider that even a delay of zero seconds takes up some overhead[39] time.

Anyway, whatever the choice of words, this time-based model make it necessary to rely on the computer timer to schedule a system update (in computer parlance, a *refresh*) at every chronon. This is tightly connected to animation, which, after all, is nothing more that geometry in motion: a computer simulation of a dynamic system implies that some graphic objects will move across the viewport to simulate actual situations at each chronon. This includes images representing world objects (possibly our actors, like Mario the penguin), moving across the viewport as they would in real life, or in imaginary worlds. All this leads, quite naturally, to rely on the power of computers, and, more specifically, on programming *loops*.

Technically speaking, loops describing system of even low complexity cannot be committed to an arithmetic sequence: this restricts the choice to a *while* loop. Said loop must take into account every possibility for the system to keep on working (which makes them *alive*; of course, we should know how to make it to *stop* working), and should also, when needed, include the intervention of an external operator.

What we are about to explore is *a whole new world*[40], which is described as *event-driven* in the trade.

The celebrated Merriam-Webster dictionary defines an event as "something that happens"; in computer science, this is exactly the same, with the restriction that we are referring to what occurs during the execution of a program. Just to make an example, user interaction is one of such events.

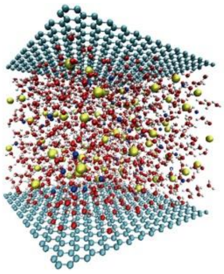
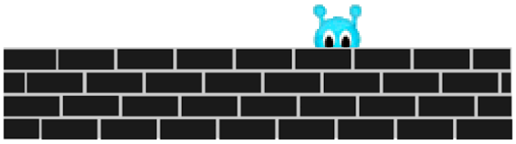
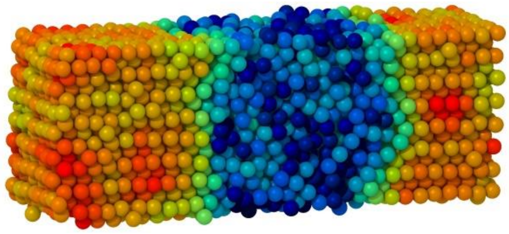
There are thousands of "things that happen" when a computer is being used, most of which are managed by the operating system. For the common user, events belong to the following categories:

- Timer (the only automated one, usually happening, as seen before, at given chronons);
- Keyboard;
- Mouse;
- Viewport commands (such as the "close" button).

[39] *Overhead* is anything the computer does that is not directly linked to doing calculations for the user, but to the internal workings. Incidentally, the computer itself is a *system* which has been programmed for a variety or tasks.

[40] From *Aladdin* (1992).

How to set up a simulation with marioGraphics

Let us then review how to upgrade our programs with the addition of an event loop.

1 A bit of user interaction

Of course, there must also be a beginning and an end to the action of our program, according to the boundary rules of the system, which may include user interaction. This means that the loop representing the development of the system must be enhanced with further considerations regarding external intervention, mostly done through keyboard or mouse action, and sometimes with timers.

All this requires our programs to be modified, including all the new considerations. Let us review the basic structure already encountered:

```
while ...
    ...             #   change the scenario
    d.draw(...)     #   draw objects (or just move them, in some situations)
```

The new structure must take some new things into account, so there will be a few changes. One good thing is using a variable for the time interval, but this is only a detail to improve timer management. The new structure should look like the following:

```
dt = ...            #   Determine a time interval
d.start()           #   Start an internal clock (ONLY when necessary)
while d.alive():    #   Check for termination (user-driven or otherwise)
    ...             #   Display the playing field
    if ...          #   Check for user interaction or other events
    d.tick(dt)      #   Pause and reset events (also: "sleep", "rest")
```

Before starting the loop, the designer can be set up with the **`start()`** method: this will void any previous action, making the designer ready for action. However, this is *not* necessary in the initial event loop (which, in most cases, will be the only one present). The designer will check for user termination by using the combination of two methods, **`tick()`** and **`alive()`**, which, in its most basic form, poll the event queue for user-driven termination, which can be obtained in two ways:

- Pressing the `Esc` key;
- Clicking on the `Close` button of the viewport.

When either action happens, the loop terminates[41] and control passes to the remainder of the program (in most cases, just to end it). The `tick` method *must* be used because it is the only one that intercepts user interaction.

The following snippet will display Mario the penguin, then wait for user interaction. We may notice that the call to the `freeze` method is not needed any more, the wait having been delegated to the `alive` loop:

```
dt = .01
img = Image('mario4.gif')
d.draw(img)
d.start()
while d.alive():
    d.tick(dt)
d.destroy()
del d
```

NOTE

Nothing happens in this program: it only waits for the user to close the viewport.

The call to **`tick()`** is necessary, or the designer would not intercept user interaction.

[41] The details of how this happens are hidden in the designer code, in the library `marioGraphics.py`.

2 Moving around

Another thing that could be changed (but not necessarily) is the way the viewport is updated. Sometimes there will be no need to erase the viewport with the `clear()` method, because objects can just be moved around or redrawn, leaving no trace of their previous position and state.

We will examine an example on how to update a couple of `Image` and `Text` objects. The same applies to other entities (after all, images and text are nothing more than glorified `Point` objects).

Such objects can be moved relative to its position or to a new position altogether; they can also be altered in place (typically, this happens with a changing label). In all these cases, it will not be necessary to issue a `clear` method (though the call to `show` must remain, if we want to check for user interaction). The corresponding methods are:

`go(x,y)`	Move **to** a new position, identified by two coordinates
`go(p)`	Move **to** a new position, identified by a `Point` object
`move(dx,dy)`	Move **by** the given distances
`move(p)`	Move **by** the coordinates of a `Point` object
`project(x,y,d)`	Move **towards** another point by a given distance
`project(p,d)`	Move **towards** a `Point` object by a given distance
`redraw()`	Replace the object, when needed (normally, after some changes)

The following example (which does not need interaction) moves an image to a new position and updates a `Text` object. Only one of the moving options is shown; adding the others is left as an exercise:

```
img = Image('mario1.gif')
tx = Text('Start at the origin',-2,2)
tx.anchor='w'              #   align to the LEFT (to the right of the anchor)
tx.draw(d,BLUE)            #   draw the two objects
img.draw(d)
d.tick(3)
tx.label='example: go(1,1)' #   change the label
tx.redraw()                #   update the Text object in the viewport
img.go(1,1)                #   update the Image object
```

3 Checking for program termination

Power is nothing without control, recited a historical commercial. We may add that brakes are equally as important as speed: there may be several ways of ending the event loop, which must be built in the program code, and not entrusted to the user (with `Esc` or `Close`). It only comes to reason that the designer has a **`stop()`** method to do just this, and it can be called under particular conditions that may arise in the state of the system during its evolution. This is called *event-driven termination*. For example, an actor could move into a trap, a timer may expire, or the user could click a specific area on the viewport: those who are familiar with videogames know these and scores of other game-ending situations.

The event loop will thus be expanded with possible further actions after updating the viewport, some of which may lead the program to terminate the simulation:

```
while g.alive():
    g.draw(...)/x.move()  #   Display new scenario (or move certain objects)
    ...                   #   Possible action
    if ...:               #   Checking the rules of the game
        g.stop()          #   System-driven termination
    ...                   #   Possible further action
    g.tick(dt)
```

In the following example, the event loop will end when Mario the penguin exits the viewport:

```
img = Image('mario1.gif')
img.draw(g)
while g.alive():
    img.move(.1,0)        #   only move the shape
    if img.x >= g.r:      #   checking boundary conditions
        g.stop()          #   ending the game by some rules
    g.tick(dt)
```

4 Putting it all together

To sum it up, the event loop management must include the following:

- Set up the designer with a call to **`start()`**, only when needed;
- start the event loop, checking the **`alive()`** condition;
- of course, the **`dead()`** method of the designer yields the opposite result of **`alive()`**, but that would hardly be needed;
- detect possible user interaction with the proper interceptors;
- have the system evolve according to its internal rules (which may include various modes of user interaction);
- show the new situation;
- add any other considerations called upon by the system rules (including termination);
- advance one chronon with **`tick()`**: this call is *necessary*, because it activates the designer interceptors; failing to do so would leave the viewport on the screen with *no user interaction possible*;
- in some cases, the simulation can be sped up[42] with a null chronon, which still uses some system time, **`Python`** not being the fastest environment around.

If needed, the program can check how the event loop has been stopped, then take the appropriate action:

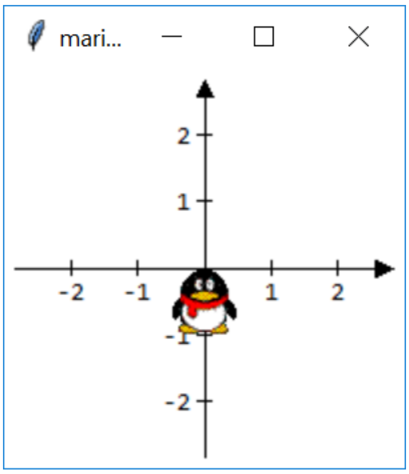
```
if g.closed()              #   terminated by Esc key or Close button
    ...                    #   some final action
if g.terminated()          #   terminated with a call to stop()
    ...                    #   some different final action
```

In the following example, we will introduce an image, and check for a mouse click on it.

```
i = Image('mario1.gif')
i.anchor = 'n'
i.draw(g)

while g.alive():

    if i.clicked():
        print('there was a click on the image')

    g.tick(.05)
```

The image anchor, which is the origin, is above it (due "north"), so Mario will lie just under the x axis.

In the event loop, a message will be issued whenever the user clicks on the image. Nothing else is due to happen, this being only a brief example of interaction.



From now on, the event loop will sometimes be only implied. This will not be a good enough reason to ignore it.

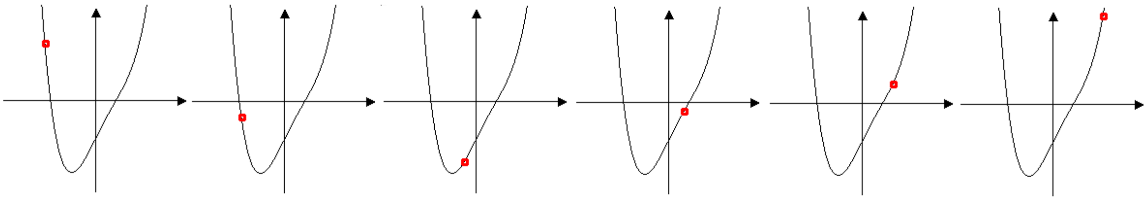
[42] Some would say "speeded up", and be criticized by language purists, but that question is borderline.

Application to calculus: tracing a function (back)

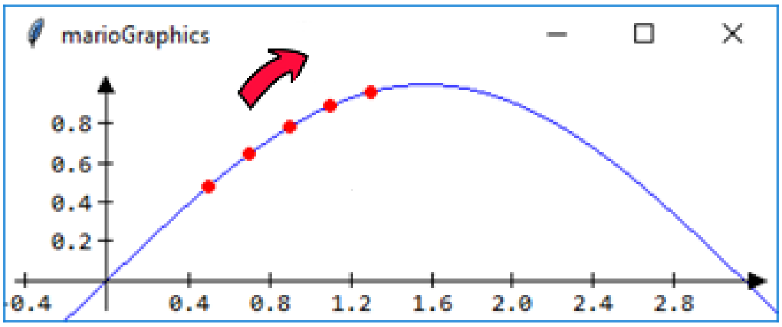
We have already met complex structures (**Polynomial**, **Rational**, and **Function** objects) quite common in calculus, as in the following example:

```
p = Polynomial(-1,2,0,-1,1)     #    y = x4 - x3 + 2x – 1
```

We will now simulate a point moving along such a curve, much as a pearl or a bead can be slid along a line.



Any function, though, can be used, as the sinusoid `f = Function("sin(x)")` of this example:



The simplest solution is using the **x** axis as a pedal line, evaluating the polynomial ordinate corresponding to each point, then drawing the points thus obtained. This is, basically, the same process by which the designer draws function graphs: the difference is that, at each step, the field must be *cleared* and *redrawn*, so as to give the illusion of the point moving. In the example of the sine function, two additional variables (`LEFT` and `RIGHT`) were set to `0` and `pi`, respectively, to allow the point to bounce back once it reaches either of them.

```
While g.alive()                   #   In the event loop
    p.y = f.y(p.x)
    g.clear()                     #   Clear, then redraw
    g.draw(f,BLUE)
    p.draw(g,RED)
    g.axes(0.5,0.4)
    p.x+=dx                       #   Move, then decide what to do
    if p.x>=RIGHT or p.x<=LEFT:
        p.x-= dx                  #   Bounce back
        dx = -dx                  #   Change direction of movement
    g.tick(0.)                    #   Shortest chronon possible
```

The problem with this solution is that **Tkinter** is not the fastest memory manager, so the chronon must be set to a ridiculously low value: and even with **0** the movement is sluggish.

In such situations, it is possible to just move the point, which is one quite nice feature of **Tkinter**. It will then suffice to draw the scenario one time, before the event loop, during which point **p** will just be moved to a new position. This can be determined by using a slack point (**q**), not shown on the viewport, acting only as the "target" position for point **p**:

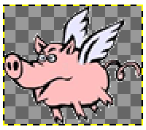
```
While g.alive()
    q.x = p.x + dx                #   Slack target point
    if LEFT <= q.x <= RIGHT:      #   Check the point against the interval
        q.y = f.y(q.x)            #   Compute the new coordinates
        p.go(q)                   #   Move "p" to the new position
    else:
        dx = -dx                  #   Bounce back
    g.tick(0.01)
```

When pigs fly: user-program interaction (back)

Thomas Johnstone “Tommy” Lawrence (1940-2018), a Scot, minded the net for Liverpool during the ‘60s, appearing in 390 games: he was nicknamed “the Flying Pig”, due to his portly figure. Sadly, he passed away in 2018. A true gentleman, on and off the pitch. Well, maybe not quite “on” the pitch, where he would often, in his own words, “bring them down”: in those days, the game was played by real men, not by tattooed sissies who drop unceremoniously to the ground whenever one farts in their same time zone.

Tommy Lawrence, the “Flying Pig”, in his playing days and in 2015.

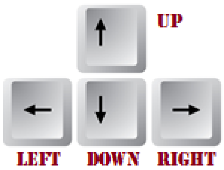
In this game we will honour[43] Tommy by flying a different kind of pig. The rules are very simple, needing only two images: one landscape for the background raster, and one winged pig as a moving sprite. The pig image should be tweaked with some software (for example, **GIMP**) to obtain a transparent contour, so as to appear a cutout rather than a rectangle:



Adding transparency with GIMP

The final result can be summarized with a pictorial expression:

In this exercise we will introduce the concept of *user interaction*, linking certain keys to action on our flying pig. Typically, sprites are moved by pressing the arrow keys:



Event-driven programming languages (as is the case with **Python**) have elements which can poll the operating system interface and capture events. These elements are called *interceptors*; basically, what they do is sit around and wait for an event, which they report to the program (more or less, like certain fighter pilots do with enemy aircraft).

Our designer, of course, has event-intercepting capabilities, which can be used in the event loop:

```
...                             #   in the event loop
    if g.key:                   #   check if a key has been pressed
        ...
    if g.click:                 #   check if the mouse has been clicked
        ...
    if img.clicked():           #   check if an image has been clicked on
        ...
```

We will now examine the **g.key** property. More on clicks soon.

[43] He was a Scot, after all: this explain the British spelling.

Keyboard interaction

(back)

Event management in **Tkinter** (and in all gaming engines, for that matter) requires polling the *event queue*, which is managed by the operating system. In this first exercise, we will only deal with keyboard events; but, as we will soon see, the other events are handled in a very similar way.

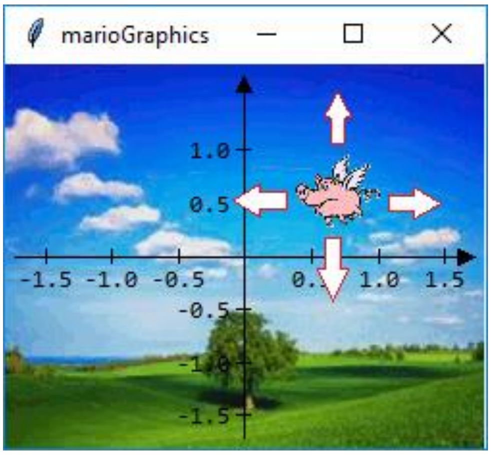
We will also add to our "flying pig" program one *timer* (or *clock*) variable, which is the standard way of controlling the flow of time (while at the same time managing the event queue).

The example shows a more or less complete procedure of event management in **marioGraphics**. Please note that arrow keys have *names*, unlike letters and other symbols identified by the actual characters.

```
from marioGraphics import *

g = Designer(250,200,-1.8,1.8,1.8,-1.8)

s1 = Image("willy.gif",0.7,0.8)            #   Willy the pig
s1.scale(0.5)                              #   scale Willy down
s2 = Image("landscape.gif")                #   A landscape
g.draw(s2)                                 #   Draw the images
g.draw(s1)
dt = 0.1                                   #   chronon

while g.alive():

    if g.key                               #   poll the keyboard event
        z = g.key                          #   test for arrow keys
        if z == 'Right':                   #   determine action to be taken
            s1.move(0.1,0)
        elif z == 'Left':
            s1.move(-0.1,0)
        elif z == 'Up':
            s1.move(0,0.1)
        elif z == 'Down':
            s1.move(0,-0.1)

    g.tick(dt)                             #   advance timer

g.destroy()
del g
```

NOTE: Property **key** of the designer is assigned to a variable only to keep the code cleaner by using a shorter name in the **if/elif** sequence.

In the complete program, the four arrow keys will send Willy flying through the air (at least, in our imagination: after all, it's *just an illusion*[44]):



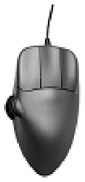
[44] *Just an illusion* is a song from the 1982 album *In the Heat of the Night*, by the British trio *Imagination*.

Mouse interaction (back)

Mouse clicks are slightly more complex to manage, having two things to consider:

- The mouse button used;
- The position of the click.

Some designer properties (**click**, **rightClick**, **middleClick**, **move**, **wheel**, **enter**, **leave**, and **mouse**) can be used to intercept the three possible mouse click. Yes, there used to be a *middle* button, now relegated to the dustbin of history:

Legacy three-button mice

Said button has been substituted by a *wheel* (which, actually, is more often *scrolled* than *clicked*). So, our list of mouse events comprises three choices, including a clumsy wheel click. Although it is possible to handle wheel scrolls with **Tkinter**, there is no necessity to do so in this course: instead, it could be interesting to intercept mouse movements (which the designer does, on request).

The main problem with intercepting the mouse position is that the click is detected in *screen* coordinates with the event interceptor: in order to get the corresponding *world* coordinates (the ones that really matter), the designer stores them as a **Point** shape.

When, for example, a mouse left click is detected, the designer initializes a tuple of integers and a **Point** as:

```
g.click[0], g.click[1]          #   click position in screen coordinates
g.mouse                         #   click position in world coordinates
```

Of course, the **Point** object `g.mouse` is totally legit, and can be treated accordingly: its properties are `g.mouse.x` and `g.mouse.y`, and we can use the familiar methods, such as `distance()` and `project()`.

When other mouse events happen, the `g.mouse` point still contains the world position of the event, while the screen coordinates are stored in other tuples, called:

```
g.rightClick                    #   right click position
g.middleClick                   #   middle click position
g.move                          #   mouse movement position
```

The following snippet tracks the mouse events on the execution windows: any one of the designer tuples can be checked for existence, then the corresponding action can be taken.

To give another example, the middle click event unusually terminates the event loop, not because it is necessary (in our example, the **Esc** key and the “close” button more than suffice), but because, in some occasions, the rules of the system dictate how to terminate the animation in different ways (clearly, nobody will be ever likely to use the “middle click” event).

```
...                             #   This is INSIDE the event loop
    if g.click or g.rightClick:     #   Left or right click
        p = Point(g.mouse)
        writeln('pig at    ',s1.x,s1.y)
        writeln('click at  ',p.x,p.y)
        writeln('distance: ',p.distance(s1),'\n')
        while s1.distance(p)>0.04:
            s1.project(p,0.04)
            g.sleep(0.01)
        s1.go(p)
    if g.middleClick:               #   Middle click: an experiment
        writeln('terminated by mouse event')
        g.stop()
```

Can’t not tweet this (back)

The self-described “tinkerer” Tim Holman (**http://tholman.com**) has, in his own words, taken “annoying social media popup crap[45] to the next level”. He has written a useless Web page[46], where a Twitter icon/link constantly (and, indeed, quite annoyingly) follows the mouse, no matter how fast it is moved, so that the only possible user action is activating a tweet dialog window (hence the slightly twisted grammar of the page address):

http://cant-not-tweet-this.com

Apart from any considerations we can make on social media, our focus is on the *behavior* of Tim’s page (which is nothing else that the effect of some clever JavaScript), noting that it can be easily reproduced with the **Tkinter** interceptors, and even more so with the designer methods.

The main idea is to assign the mouse coordinates to a given image (of course, it will feature Mario the penguin). We will apply some changes to the mechanism seen before:

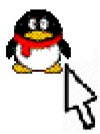
- finding the mouse position is an integral part of the event loop, so it will be intercepted there;
- we will use the **mouse** point of thedesigner method to get (what else) the mouse coordinates;

Said method uses the **Tkinter** mouse interceptor, then translates the mouse coordinates according to the current calibration, returning them as a **Point** object.

```
mario = Image("Mario1.gif")
...
while g.alive():
    if g.move:
        mario.go(g.mouse)
    g.tick(0.01)
```

A slightly different solution, found on **www.processing.org** , adds a slowing mechanism to the object following the pointer. The key, in this case, is keeping track of the previous position of the object, then cut the distance to the new mouse pointer position by a small fraction, so as to give the impression of the object slowing down as it approaches the pointer. We will also use a different method, which moves the shape by given **dx** and **dy** amounts, rather than addressing a specific point.

```
...
delta = 0.5
while g.alive():
    if g.move:
        dx = g.mouse.x - mario.x
        dy = g.mouse.y - mario.y
        mario.move(dx*delta, dy*delta)
    g.tick(0.1)
```



This could also be done with a call to the **project** method:

```
...
    if g.move:
        mario.project(g.mouse,mario.distance(mouse)*delta)
```

Of course, there are many variations on this theme, and the limit is only the fantasy of the programmer.

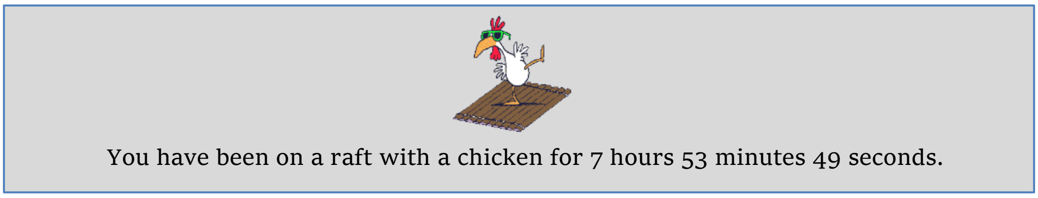
---

45 Crap = solid, organic, animal waste.

46 The Web hosts several collections of “useless pages”, which have no particular purpose, and appear to having been written just for fun. This particular page appears in many such lists.

Timer management (back)

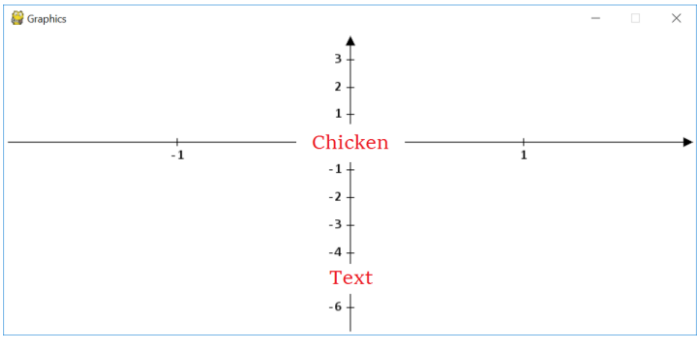
Having mentioned useless Web pages, this example is a homage paid to the immortal “Chicken on a raft”, which is nothing else than a static image with a running timer, and the homonymous song[47] playing continuously. This image has been obtained after a while (the actual background is black):



http://chickenonaraft.com

Of course, this exercise in futility is an excuse to analyze in some detail how to manage the timer with both **Python** and **Tkinter**. We will also review how to display text in the viewport with a **Text** shape.

First, let us plan the geometry. The calibration tries to match the original page; the chicken is placed at **(0,0)**, which are the default coordinates; the **Text** object, which is analogous to a point, is placed at **(0,-5)**, and its property will report the time of activity (so to say) of the simulation.



The timer management has two aspects: the calculation of the elapsed time, which can be managed with **Python** functions, and the pace of the event loop, which needs the usual designer **tick** method. The timer interval, of course, will be set to **1** second, to match the behavior of the Web page. The initial time will be stored into a variable named **then**, and the changing time into another variable named **now**: being obtained in seconds, the difference must be split into several components (days, hours, minutes, seconds), with a sequence of divisions and remainders. The Python functions best suited to manage this simulation are **gmtime** (**G**reenwich **m**ean **t**ime) and **mktime**: the former returns the date and time as a 9-item tuple (including date items), the latter transforms such a tuple into a numeric value in milliseconds.

The snippet shows how to calculate the total number of seconds at each tick of the clock (**now-then**), cast to an integer. To obtain days, hours, minutes, and seconds, this number can be processed with divisions and remainders, remembering that: one day contains **86400** seconds, one hour contains **3600** seconds, one minute contains **60** seconds. The calculations and the preparation of the text are left as exercises.

```
g = Designer(800,350,-2,4,2,-6.8)
g.background(BLACK)
chicken = Image("clyde.gif")
t = Text(0,-5)
g.draw(chicken)
g.draw(t,WHITE)
then = gmtime()
g.start(1)
while g.alive():
    now = gmtime()
    seconds = int(mktime(now) - mktime(then))
    ...                     #   compute days, hours, minutes, seconds
    t.label = ...           #   prepare text
    t.redraw(g)             #   display text
    g.tick(1)               #   timer (in seconds)
```

> In this example, the event loop is started with a one-second delay, to show the initial situation before the first timer update. An alternate solution could be placing the **tick()** call at the beginning of the event loop, but the masters advice against doing so.

---

[47] *Chicken on a raft*, here in a version by Tim Lewis, is an actual song, composed by Cyril Tawney (1930-2005) in the tradition of maritime shanties (work songs). It refers to a dish of fried egg on fried bread, which was apparently not popular among the Royal Navy seamen (the *limeys*). Lyrics and music can be found, among various sites, at:

http://mudcat.org/@displaysong.cfm?SongID=1159

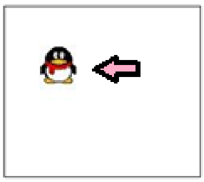
The drunkard's walk (back)

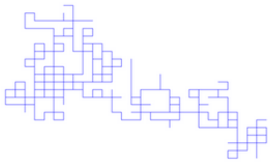
One of the classic simulation problems deals with a random movement, in some ways similar to the jerky and shaky walk characterizing a drunk person. There are many educational Web sites dealing with this puzzle, a couple of which are the following:

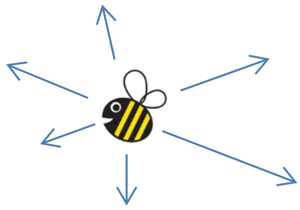
https://www.quantamagazine.org/20160818-random-walk-puzzle/
http://www.math.cornell.edu/~mec/Winter2009/Thompson/randomwalks.html

There are two simulation methods: one consists in randomly moving a geometric shape in the viewport (normally an image, but a point or small circle would work just the same), the other in drawing the segments along which the movement is done (without drawing the moving shape). Of course, tracking the drunkard during the movement requires keeping all the segments visible, without clearing the viewport.

From the point of view of physics, the limit of the random walk is a Brownian motion: for this reason, while Mario the penguin is the perfect choice for slow simulations, fast and unpredictable ones will be best rendered with Melanie the bee[48]:







There also are different ways to determine the new position of the drunkard, among which:

1. one step away along any of four directions, parallel to the coordinate axes
2. one step away at an angle from the horizontal axis, thus landing on the surrounding *circle*;
3. to a random position in the surrounding *disk*.

While the first type of motion can easily be solved by altering any of the coordinates of the actor (as in the example of the flying pig), in the other cases it may be best to use a segment, which can be rotated and stretched to follow the events.

The length of a standard step depends on the drunkard. A penguin step is about **15** cm; a human step is about **40**; a bee dart can vary anywhere from a few centimeters to a couple of meters. The program may contain an initialization of the chosen value (which in the most general case would be an upper limit):

```
x = 0.15
```

Determining the movement should follow the chosen method:

Parallel to one axis	`m = randrange(4)` `m = randint(0,3)`	The two calls are equivalent; then move the drunkard in **4** possible ways
Choosing a random angle	`a = uniform(0,2*pi)`	Between **0** and **2*pi**
Choosing a random step	`step = x * random()`	Between **0** and one full step or dash

Reviewing the random capabilities of **Python**:

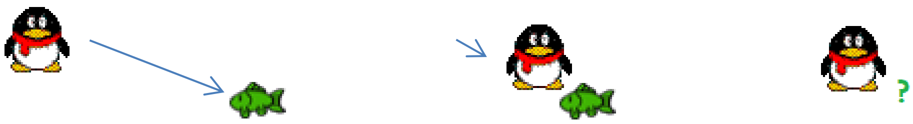
- **randrange(n)** yields an integer random number between **0** and **n-1**;
- **randint(a,b)** yields an integer random number between **a** and **b**;
- **uniform(a,b)** yields a double-precision random number in the interval **a<=number<b**;
- **random()** yields a double-precision random number in the interval **0<=number<1**.

[48] Beehives are often cited as an example of Brownian motion, since bees dart around in seemingly random bursts.

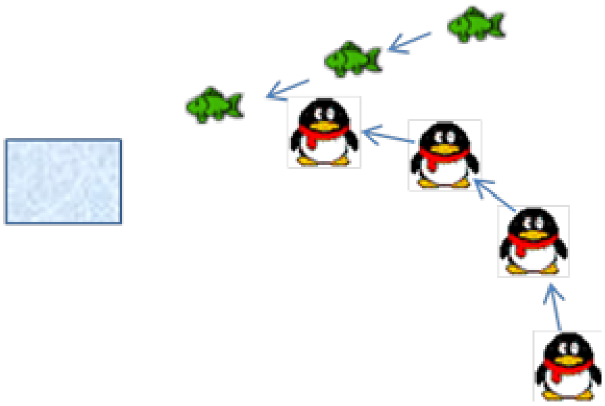
Mario and Sushi (or: the chase) (back)

This classic chase game has appeared in many guises, including one featuring a cowardly gladiator trying to escape from a lion. In our version, Mario the penguin will try to reach Sushi the fish to have a quick meal.

The simplest solution to this problem features a still Sushi with Mario approaching him via the shortest path (which is, understandably, a straight line). The game ends when the *distance*[49] between them decreases under a given threshold, after which Sushi should quietly disappear (presumably, into Mario's belly):



To carry the simulation a step further, we may give Sushi a chance, adding a safe den that he will try to reach, before being himself reached (and presumably eaten) by Mario. In this version, the shortest path for Sushi would be a straight line, while Mario could engage in a curvilinear path, due to the motion of his target (at each chronon, Mario will have to adjust his trajectory towards a moving Sushi). Of course, if Mario knew better, he would point straight to the den, thus shortening his path, but penguins are not renowned for their brains, so the chase will proceed more or less like in the following image, with the den represented by a sort of fish tank:



The rules of the game can be summarized as follows:

- Sushi moves towards the den (geometrically, its center);
- Mario moves towards Sushi;
- if Sushi reaches his den with Mario far enough, the game ends in his favor (no meal);
- if Mario reaches Sushi outside the den, the game ends in his favor (meal).

After defining the den as a rectangle, its center can be found with the corresponding method:

```
den = Rectangle(...)
center = den.center()                #   returns a Point object
```

In the simplest solution, the event loop just moves the two actors, whose speed (which is actually the step) is better represented with variables (`dSushi` and `dMario`) than with constants, so as to simplify possible modifications:

```
sushi = sushi.project( center, dSushi)
mario = mario.project( sushi, dMario)
```

It would also be a good choice to avoid Mario entering the den, with a condition:

```
if not mario.ping(sushi,dMario).inside(den):
    mario = mario.projection(sushi,dMario)
```

The other thing to decide is how to start the game, choosing the positions of the three shapes either interactively or at random. This is left to the fantasy of the programmer, along with possible user intervention, which may include cheating in favor of either actor.

[49] Once again, a *geometric* concept is a key factor in a simulation problem.

The race of the penguins (back)

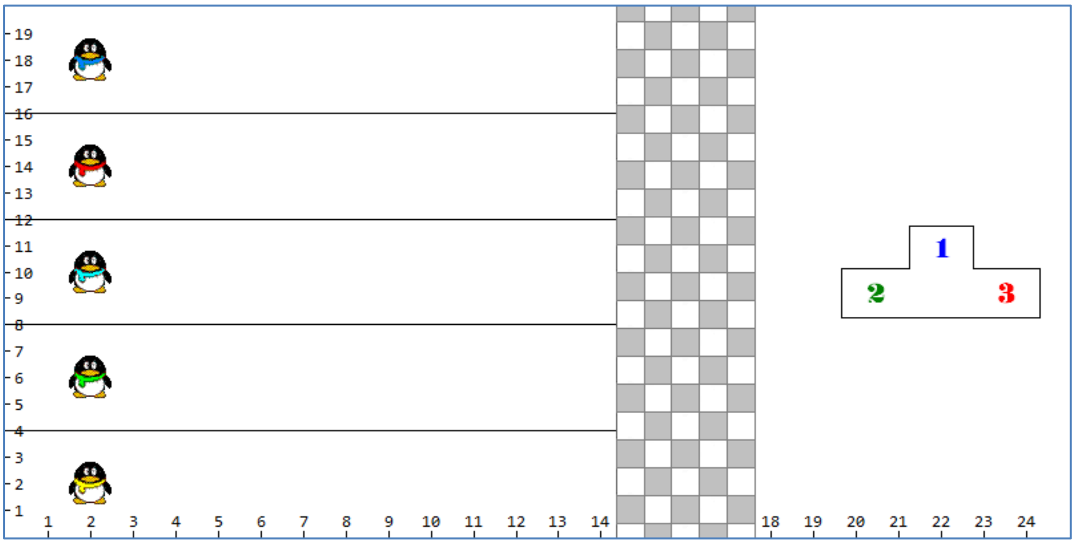
This is a classic game which appeared (as *horserace*) shortly after the introduction of video displays (that was back in the early 1980s). As the name implies, some horses were pitted on a linear track, spanning the width of the display, then moved at random until all of them had arrived at the finish line.

This version is only slightly less crude than the original one, using a very simplified race track, and featuring the ubiquitous "Mario the penguin", in five different avatars. The racers are depicted in a quintet of images:

Mario0.GIF Mario1.GIF Mario2.GIF Mario3.GIF Mario4.GIF

The playing field needs just a couple of other pictures (a checkered finish line and a podium for the three medalists: `finish.bmp` and `podium.bmp`); for the lanes, one segment will suffice, with multiple displays. In the example (where the world is `0,20,25,0`), the race takes place between `2` and `16` units, while the podium is just to the right of the finish line (at `10,22`). The displacements of the three medalists with respect to the podium are, respectively, `(0,2.65)`, `(-1.5,1)`, and `(1.5,1)`.

Note: a simpler version of the game can skip medalists and podium.



The event loop is quite simplified, lacking user interaction and only checking for the timer. The game, though, must provide for the advancement of the runners and for the final classification:

```
mario = [None for x in range(5)]          #   Using an array of images
for j in range(len(mario)):
    Mario[j] = Image( "mario"+str(j)+'.gif', 2, 14 - 4 * (j-1))
...
while g.alive():
    j = randrange(n)                      #   Randomly choose who is moving
    if Mario[j].x < finish:               #   Only if not arrived
        Mario[j].x+=g.px*speed;
        if Mario[j].x >= finish:          #   Arrived?
            ...
            if ...                        #   Test if medal available
                medalists[medal]=Mario[j] #   Add to medalists
```

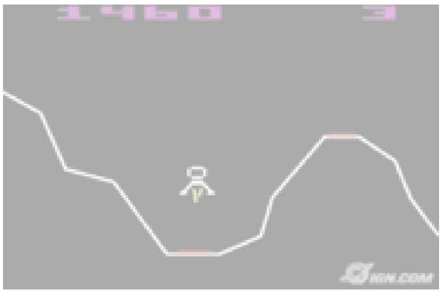
At the end of the race, some ceremony can take place, showing the medalists on the podium and a big smiley by the winner (with the ordinate corresponding to the racer).

```
winner = Image("smiley2.jpg",finish+2.5,0) #   at the beginning
...
if winner.y:                               #   at the end
    g.draw(winner,0.4)
```

Lunar Lander (back)

Lunar Lander (also called **Moon Lander**) is an early computer game written in 1973 on a commission from the now-defunct **DEC** (**D**igital **E**quipment **C**orporation). They used it to demonstrate the capabilities of their new **GT40** graphic terminal in several trade shows and conferences.

Of course, state-of-the-art terminals in the early '70s were a far cry from today's equipment, offering a crude monochromatic display, but they were the only option, and a costly one at that[50].



The goal of the game was to safely land a lunar module on the surface of the moon, slowing down the fall with the thrust of a retrorocket: too high a landing speed caused the vehicle to crash; too strong a thrust could send it flying off into space. Unsuccessful pilots were taunted by the program for their ineptitude.

The game was also produced in an arcade version, which kept the sober appearance of the playing field: a black background showing the outline of the moon's surface and a crude image of the lander.

The rules of the game are quite straightforward:

- The initial height and the available fuel are set at random;
- Gravity on the Moon is about **1/6** that of the Earth: g_{moon} **= 1.6 m/sec**^{2};
- The lander has a limited amount of fuel, whose consumption is regulated by the pilot;
- One unit of fuel accelerates the ship upward by **10 cm/sec**^{2};
- The maximum number of units of fuel per second is **50**, corresponding to **5 m/sec**^{2};
- The safe landing speed is anything up to **2 m/sec**;
- Of course, running out of fuel will send the vehicle into free fall.

Just like with the flying pig simulation, background and calibration should match, so as to fill up the viewport with a landscape (or, rather, a *moonscape*). A **100x500** image is available, along with a **45x29** alien spaceship.

We will set up an event loop, with some user interaction:

```
if g.key                    #   polling the keyboard event
    z = g.key
    if z == 'Up':           #    Up arrow: increase fuel intake
        ...
    if z == 'Down':         #    Down arrow: decrease fuel intake
        ...
```

To apply the laws of physics, simulations call for repeated calculations, where the speed of the vehicle is adjusted every given unit of time, according to the current acceleration, which, as immortalized in Newton's Second Law of Motion, is proportional to the total force acting on a body (in this case, the combination of gravitation and rocket propulsion).

In short, what many simulation programs do is computing significant data from past values, a procedure called *difference equation* method, which is activated at each chronon. As always, the value of a chronon depends on the scale of events: in this model, we can safely set it at one tenth of a second.

So, after adjusting for the fuel consumption, the program can compute the data it needs for determining the motion of the spacecraft:

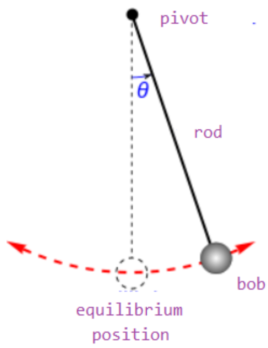
```
v = v + a * Δt          #    speed
y = y + v * Δt          #    distance from surface
```

Of course, the value of **y** should be used in the boundary rules and in the conditions of the event loop.

---

[50] When launching the terminal into the market in 1972, **DEC** made a great deal about the selling price of "under $11,000". A typical car cost around $4,000, a home around $40,000, and a family could live for a year on $14,000.

On pendulums, and more about physics (back)



Galileo Galilei (1564-1642) had quite a problem with the Church, as his trial for heresy shows. But even as a young man he may have had some issues, at least with ceremony: his biographer Vincenzo Viviani wrote that Galileo discovered the properties of pendulums observing a swaying chandelier while attending Mass, presumably a little bored by the sermon.

As is well known from physics, an ideal pendulum comprises three elements:

- one frictionless pivot;
- one massless rod;
- one massive bob.

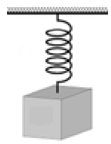
The oscillations of an ideal pendulum around its equilibrium position are *isochronous* (that is, they have the same duration), at least when the amplitude of its swing are small[51], and depend only on the length of the rod. For this reason, pendulums found a universal use in clocks[52]. The formula for the period **T** is the well-known $T = 2\pi\sqrt{L/g}$ (also without mass).

The difference method for our simulation calls for the value of the instantaneous acceleration. We will use the component of gravity which is tangential to the motion of the pendulum (the perpendicular one is canceled by the static action of the rod). Two estimates could be used (and compared for accuracy with the analytic formulas), since in small angles the sine can be approximated by the arc (after all, $\lim_{x\to 0}\frac{sin(x)}{x} = 1$).

Estimate 1: `a = - g * sin(x)` **Estimate 2:** `a = - g * x`

As usual, the computer simulation is quite simple: what we need is a segment for the rod and a circle (or disk, or ring) for the bob (additionally, a dot can be used to mark the pivot).

Wait, there's more

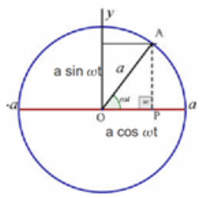


The pendulum undergoes a *simple harmonic motion*, that is, a periodic motion where the restoring force is both directly proportional and opposite to the displacement. But in physics, there is plenty of harmonic oscillators. One such system is the *spring*, which obeys Hooke's law[53]:

`F = - k * x`

where **k** is the elastic constant of the spring. There are multiple analogies with the pendulum: for example, the mass of the spring should be negligible with respect to that of the weight suspended to it; also, there is a limit to the amplitude of the spring extension. The latter, however, depends on physical, rather than geometric, properties: the harmonic limit for a spring is the displacement at which the spring is so stretched that it becomes permanently deformed, thus losing its restoring properties.

A link with geometry



Linear harmonic motion is inexorably linked to geometry through trigonometry: in fact, when a point travels along a circle with constant angular speed, its projection on any line moves harmonically. This can easily be shown with programs similar to previous ones with *circular pedal lines*.

With a little help from trigonometry (as shown in the image from **astarmathsandphysics.com**), the general equation of harmonic motions, as a function of time **t**, is:

`x = A * cos(ωt+γ)` (where `A=radius`; `ω=angular speed`; `γ=phase`)

---

[51] Just *how* small is *small*? Textbooks seem not to care about giving an *actual* figure, reinforcing the suspect that most of them are more or less copied from others. A safe measure is no more than a tenth of a radian. Prof. Francesio gave yours truly the slightly smaller value of **4°** (that is, **0.07** radians).

[52] The elongated shape of grandfather clocks is a clear reminder of the fact that isochronicity requires *small* oscillation amplitudes.

[53] Robert Hooke (1635-1703), a fine scientist based in Oxford, and a contemporary of Isaac Newton, first published his result in 1675 as an anagram (with all the letters in alphabetical order: ***ceiiinosssttuv***). Later (in 1678) he revealed the solution, which reads ***ut tensio, sic vis*** (as the extension, so the force).

Further adventures in calculus (back)

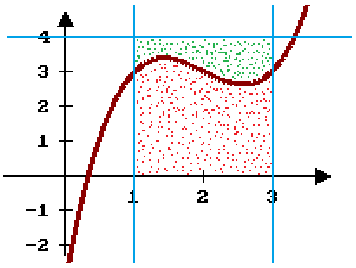
The definite integral

Integral calculus was born from the problem of finding areas delimited by curves. In the example below, the given curve (which could be any among **`Polynomial`**, **`Rational`**, and **`Function`**) is:

$$y = x^3 - 6x^2 + 11x - 3$$

The problem is finding the area of the shape delimited by the curve, the x-axis, and two vertical lines at given abscissas (say, **`1`** and **`3`**). A classic Monte Carlo method for finding said area is quite similar to the one adopted for finding the value of **π**, also relying on a proportion.

The solution requires finding a rectangle large enough to contain the shape, then generate a cloud of points belonging to the rectangle. Some points will fall below or on the curve (**red**), some above (**green**): the ratio of red points to the total number of points is the approximation of the ratio of the unknown area to the area of the rectangle.



Finding the area with calculus

The fundamental theorems of integral calculus ensure that the same area can be found by using a *primitive* function of the given one. Such a function is also called an *antiderivative*, since its first derivative is the given function. Subtracting the values of the primitive taken at the delimiting points yields the desired area.

```
y = f(x)            given function
Y = F(x)            primitive
Area = F(b) - F(a)
```

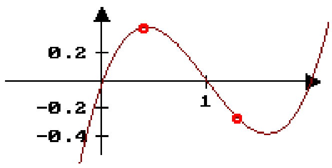
> **Finding a primitive with polynomials:**
>
> $F(x^n) = 1/(n+1) * x^{n+1}$
>
> `q.x[i]=1/i*p.x[i-1]`

With *polynomials*, it is quite simple to find the primitive algebraically, reversing the process used for finding the derivative. Once again, the analytic solution can be compared with the numerical one.

Finding a root of a function

The values of **`x`** for which the values a function **`f(x)`** is zero are called *roots* (and sometimes *zeros*). Leaving aside the fundamental theorem of algebra, let us consider Bolzano's theorem[54], which states that, if a continuous function assumes values opposite in sign at the ends of an interval, then it has at least one root in said interval.

For example, the following function has a positive value at **`0.4`**, and a negative one at **`1.3`**:, so there is at least one zero in the interval **`[0.4,1.3]`**:

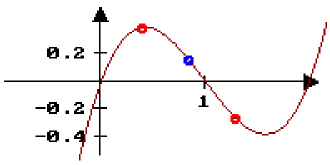


$$y = x^3 - 3x^2 + 2x$$

The so-called *bisection method* calls for evaluating the function at midpoint: if it is not zero, then a new interval can be defined, stretching from midpoint to the extreme where the function has the opposite sign. The same considerations can be applied repeatedly, finding increasingly shorter intervals, and stopping when a small enough value of the function (or a small enough *interval*) is found.

Proceeding with the example:

```
f(0.4)=0.384; f(1.3)=-0.273; f(0.85)=0.146625
```



The function is positive at midpoint, so the new interval to consider is on the right side, ranging from **`0.85`** to **`1.3`**. By repeating the process, the intervals will close around the root existing at **`1`**.

---

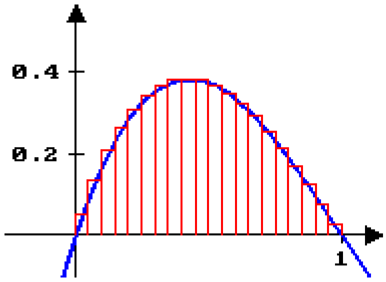
[54] Bernard Placidus Johann Nepomuk Bolzano (1781-1848) was a Bohemian priest of Italian origin.

The rectangle rule

One classic method for calculating the area delimited by a curve relies on geometric means. In the example below, the given curve is:

$$y = x^3 - 3x^2 + 2x$$

The very definition of the definite integral calls for this method. The area delimited by the graph of the function is approximated with a sequence of rectangles, spanning from the x-axis to the curve:



The width of each rectangle is a generic interval Δx, and the height is the value of the function in a point that can be chosen as:

- the starting point of the interval;
- the end point of the interval;
- the midpoint of the interval.

So, the total area is a sum of the areas of the rectangles, or:

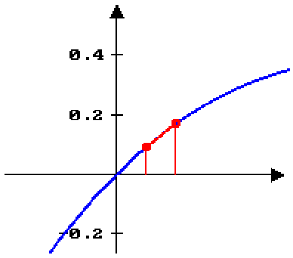
$$\sum f(x)\ \Delta x$$

When a *small enough* Δx interval is chosen, the sum approaches indefinitely the actual area. When the interval is infinitesimal, it can be written as a differential **dx**, and the symbol of the sum is substituted by a sort of elongated **S**, as in:

$$\lim_{\Delta x \to 0} \sum f(x)\ \Delta x = \int f(x)\ dx$$

As usual, the trick is the "small enough" clause. As usual, the more intervals we find, the better the approximation.

One refinement of this method is called the *trapezoid rule*, since it substitutes the top side of the rectangle with a line connecting the two points of the curve. The two bases of the trapezoid (which is sideways) are the values of the function at the end points of the interval; its height is the interval itself. The area is, of course, **(B+b)*h/2**.



Horner's method

When evaluating a polynomial in one point, the masters suggest a better method than evaluating all the powers of **x**. The so-called Horner's method only uses multiplications, thus saving precious computer resources[55]. In algebraic form, the formula can be exemplified with one case:

$$P_5(x)=((((a_5x+a_4)x+a_3)x+a_2)x+a_1)x+a_0$$

Horner and Ruffini

---

[55] It was hardly designed for computers, though, as Horner lived from 1786 to 1837. The method was also known to Paolo Ruffini (1765-1822), of polynomial fame, and the Chinese Qin Jushao (ca.1202-1261).

A study on *proxemics* (back)

Simply put, proxemics is a social science studying nonverbal interaction (both animal and human): its name comes from the Latin *proximus*, meaning *next* (and also *close*). Basically, it can be described as the study of *social distances*, that is, how people (or animals, as anybody who owns a pet could attest) feel more or less comfortable at certain distances from one another.

The American anthropologist Edward T.Hall (1914-2009), the founder of proxemics, classified human social distances[56] into four categories, with two subcategories each, related to how near one person would like to be to another one (or how *far*, if the feeling is one of distaste):

Category	Near	Far
Intimate	<=15 cm	<=45 cm
Personal	<=75 cm	<=1.2 m
Social	<=2.1 m	<=3.7 m
Public	<=7.6 m	>7.6 m

Of course, Hall used feet and inches, so the conversion to metric is approximated; also, these distances should be considered loosely, just to give an idea of interaction.

The classic simulation game, as proposed by the late, great Martin Gardner (1914-2010), deploys a certain number of hosts joining a party, such as the group of young people pictured above. The preferences can be stored into a square matrix, where each row contains representing the desired social distances of the participant with respect to the others. Quite obviously, the diagonal of the matrix must be null (people cannot stay away from themselves). A partial situation could be the following:

	Amy	Ben	Chloe	Dave	...
Amy	0	3.0	5.0	0.4	...
Ben	0.1	0	7.0	4.0	...
Chloe	5.0	0.1	0	2.0	...
Dave	0.1	4.0	0.1	0	...
...	...	...	...	...	0

It appears that Amy and Dave are friends, but Dave has a further interest on Amy; Ben and Dave do not like each other, but they share the same interest for Amy; Chloe is not friends with Amy, and is interested on Ben, who does not like her at all.

The actual result of this party is ridiculous: Amy, followed by Ben, himself followed by Chloe, circle around Dave, who only moves a couple of steps from time to time, either to avoid Ben or to approach Amy. Of course, the situation will become more complex as we add other guests.

When Gardner's model was published in *Scientific American*, computer graphics was for a selected few, so the proposed solution used a matrix of `char`, where a zone of the party hall contained the refreshment table (thus adding one's dietary choices to the game). We can avoid this supplemental problem.

Gardner's model

```
.....................
..................3..
.....1..TTTTT........
........TTTTT...4....
........TTTTT........
.2...................
.....................
```

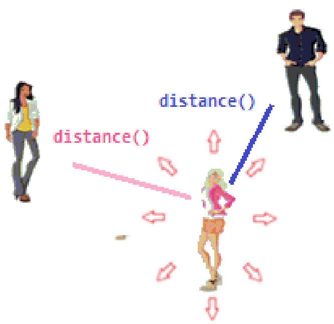
With the designer, we can use `Image` objects to represent the guests. After declaring an empty array, we can populate it with our actors:

```
guests = []
guests.append( Image("Gwen.GIF"))    #   for each guest
```

We can measure Gwen's current distress as the sum of the differences between the actual and desired distances from each of the other guests. For every given value of `i` (`i` being `6` for Gwen), a `j` loop on the other guests could sum:

```
d+= abs( guest[i].distance(guest[j]) - distances[i][j])
```

One step is about **40** cm. We can designate an array of new positions, one step away from Gwen, at various angles, calculate the distress that she would have in each one, then move to the position with the minimum distress. Note that her own position should be included in the search, because she could also stay put.



The simulation consists in extending the process to all the guests, then repeating at will.

[56] The study of *animal* proxemics was founded by the Swiss Heini Hediger (1908-1992).

Solving a linear system

One typical numeric method is the solution of a linear system of **n** equations in **n** variables (or *unknowns*). The following is an example of a **3x3** system:

```
 x – y +  z  = 4
3x + y - 2z = 1
2x + y –  z = 2
```

The system can be expressed in *matrix form*., starting from the general formulae:

$$a_{11}x_1 + a_{12}x_2 + \dots + a_{1n}x_n = b_1$$
$$a_{21}x_1 + a_{22}x_2 + \dots + a_{2n}x_n = b_2$$
$$\dots$$
$$a_{n1}x_1 + a_{n2}x_2 + \dots + a_{nn}x_n = b_n$$

Without analyzing the details, we can notice that the coefficients of the unknowns can be arranged in a square **n*n** matrix, which, with the addition of a column for the constant terms, becomes an *augmented* **n*(n+1)** matrix. Following the example, the two matrices are, respectively, **3*3** and **3*4**:

Coefficient matrix	Augmented matrix
1 -1 1 3 1 -2 2 1 -1	1 -1 1 4 3 1 -2 1 2 1 -1 2

The classic computer solution is called Gaussian elimination[57], which works by processing the augmented matrix of coefficients.

The first step consists in finding the largest (in absolute value) coefficient of the **x** unknowns, and swapping the corresponding equation with the first. This ensures a better stability of the numerical method. In our example, the system is transformed into:

System	Augmented matrix
3x + y - 2z = 1 x – y + z = 4 2x + y – z = 2	3 1 -2 1 1 -1 1 4 2 1 -1 2

The new element in the **[1,1]** position (on the diagonal) is called the *pivot*, it is the fulcrum of the following transformation, which consists in deleting the **x** unknown from all the equations, from the second one down. Of course, just deleting the coefficients is not the way: the key algebraic concept is the *linear combination* of equations. One theorem of linear algebra states that adding one equation to another yields an equivalent system (namely, one with the same solutions). This is valid also if either of the equations is multiplied by a non-zero number.

Suppose that the first equation, multiplied by 5, were added to the third equation. The resulting system would be equivalent, but would actually be more complex (not a smart choice):

```
 3x +  y -   2z = 1
  x –  y +   z  = 4
17x + 6y – 11z = 7
```

A better choice of coefficients, involving the pivot, would be:

- add to equation **2** equation **1** multiplied by **-1/3**;
- add to equation **3** equation **1** multiplied by **-2/3**.

In short, each equation is added the first one, multiplied by its first coefficient over the pivot (with a minus sign). It is evident that the new equations will have canceled the **x** term.

---

[57] Some credit also the Polish astronomer and mathematician Taddeus Banachiewicz (1882-1954). Carl Friedrich Gauss (1777-1855), of course, was the *Prince of Mathematicians*. A method similar to the Gauss-Banachiewicz reduction is called the Gauss-Jordan reduction, after the French mathematician Camille Jordan (1838-1922).

Consider what happens in the combination:

```
x - y + z + (-1/3) * ( 3x + y - 2z ) = 4 + (-1/3) * 1      (equation 2)
2x + y - z + (-2/3) * ( 3x + y - 2z ) = 2 + (-1/3) * 1     (equation 3)
```

The choices of coefficients delete the **x** unknowns from each equation below equation **1**:

```
x - x ... = ...          (equation 2)
2x - 2x ... = ...        (equation 3)
```

Of course, there is a price to pay: the remaining terms of said equations are all modified, as follows:

```
3x +     y -    2 z = 1
0x -4/3 y + 5/3 z = 11/3
0x +1/3 y + 1/3 z = 4/3
```

The procedure is easily transferred to the coefficient matrix, if we consider that the combination involves terms associated to the same unknown. For example, the new coefficient for the **z** unknown in the third equation (in **red**) is obtained by the formula **-1+(-2/3)*(-2)**, where the four numbers are, respectively:

- the **z** coefficient of equation **3** (the term to be modified);
- the **x** coefficient of equation **3** (in the same column as the pivot);
- the **x** coefficient of equation **1** (the pivot);
- the **z** coefficient of equation **1** (in the same column as term to be modified).

For this reason, the procedure is called the *rectangle rule*, since the four terms are laid out in a rectangular shape (highlighted in **red**). After combining all the equations (including the constant values), the first column of the coefficient matrix contains all zeroes below the pivot, as follows:

Before	After
3 1 -2 1	3 1 -2 1
1 -1 1 4	0 -4/3 5/3 11/3
2 1 -1 2	0 1/3 1/3 4/3

The next step may forsake equation **1**, and concentrate on the remaining two, which constitute a system of two equations in two unknowns:

```
-4/3 y + 5/3 z = 11/3          -4/3 5/3 11/3
+1/3 y + 1/3 z = 4/3            1/3 1/3  4/3
```

The pivot is already in place (in position **2,2**); the Gaussian elimination requires multiplying equation **2** by **1/4** (obtained dividing **1/3** by **-4/3** and changing the sign), then adding it to equation **3**:

```
1/3 y + 1/3 z + (1/4) * ( -4/3 y + 5/3 z ) = 4/3 + (1/4) * 11/3
```

After this, it is the second column of the coefficients matrix to contain only zeroes below the pivot. The equivalent system is called *triangular* (for obvious reasons), where the matrix is said to be *triangular*, or in *echelon form*:

System	Augmented matrix
3x + y - 2 z = 1	3 1 -2 1
- 4/3 y + 5/3 z = 11/3	0 -4/3 5/3 11/3
3/4 z = 9/4	0 0 3/4 9/4

The example stops here, as there are only three equations. In the general form, the equivalent system is just larger, but still in triangular form:

$$
\begin{array}{rcl}
a_{11}x_1 + a_{12}x_2 + \dots + a_{1n}x_n & = & b_1 \\
a_{22}x_2 + \dots + a_{2n}x_n & = & b_2 \\
\dots & & \\
a_{nn}x_n & = & b_n
\end{array}
$$

The system in triangular form can be readily solved starting from the last equation, which contains only the third unknown `z`. Note that both `a[3,4]` and `a[3,3]` are divided by `a[3,3]`, so as to represent the unknown with coefficient `1`.

```
z = (9/4) / (3/4)
z = 3
```

The two insets show the coefficient matrix before and after finding the value for `z`:

```
3    1  -2    1
0 -4/3 5/3 11/3
0    0 3/4  9/4
```

```
3    1  -2    1
0 -4/3 5/3 11/3
0    0   1    3
```

The next phase is called *back-substitution*, since the value for `z` can be plugged into the preceding equations, disposing of the `z` variable. Note that the term, once evaluated, is transported to the right-hand side of each equation and summed to the constant term.

```
3x +     y -   2  z = 1          Equation 1
    -4/3 y + 5/3  z = 11/3       Equation 2
                  z = 3          Equation 3

3x +     y -   2  * 3 = 1        Equation 1
    -4/3 y + 5/3  * 3 = 11/3     Equation 2

3x +     y = 7                   Equation 1
    -4/3 y = -4/3                Equation 2
```

With the disappearance of the `z` unknown, the second equation now contains only the `y` unknown, which can be found by dividing `a[2,3]` and `a[2,2]` by `a[2,2]`:

```
-4/3 y = -4/3                    Equation 2
y = 1
```

The three insets show the coefficient matrix before, during, and after the search of the value for `y`:

```
3    1  -2    1
0 -4/3 5/3 11/3
0    0   1    3
```

```
3    1   0    7
0 -4/3   0 -4/3
0    0   1    3
```

```
3    1   0    7
0    1   0    1
0    0   1    3
```

A new back-substitution is done on the remaining equation for the `x` unknown:

```
3x + y = 7                       Equation 1
3x + 1 = 7
3x = 6
x = 2
```

As usual, we can take a look at the matrix before, during, and after the search of the value for `x`:

```
3    1   0    7
0    1   0    1
0    0   1    3
```

```
3    0   0    6
0    1   0    1
0    0   1    3
```

```
1    0   0    2
0    1   0    1
0    0   1    3
```

The system and the coefficient matrix now are in the simplest equivalent form, yielding the solution:

```
x = 2          1  0  0  2
y = 1          0  1  0  1
z = 3          0  0  1  3
```

This type of matrix is called, for obvious reasons, *diagonal*. Incidentally, the Gauss-Jordan method extend the combination of equations to all the rows of the matrix, so as to obtain directly the diagonal form instead of the triangular one.

Refer to Appendix C for a Python solution to this problem.

Logo of `cell-auto.com`

Systems Turing machines

What, if anything, is a *cellular automaton?*

“To play *life* you must have a fairly large checkerboard and a plentiful supply of flat counters of two colors. It is possible to work with pencil and graph paper but it is much easier, particularly for beginners, to use counters and a board.”

Martin Gardner (1914-2010), *Scientific American*, October 1970

Cellular automata[58] are the classical example of *dynamic* system with many separate, interacting components, each having a finite number of states, that *vary over time*.

A cellular automaton is a model of a system of “cell” objects with the following characteristics:

- The cells live on a ***grid*** (typically in two dimensions, corresponding to a matrix of numbers);
- The dimensions of the grid and the states of the cells depend on the rules governing the system;
- Each cell has a ***state***; the number of state possibilities is finite, with the simplest models using just two values, **0** and **1** (sometimes referred to as “off” and “on”, or “dead” and “alive”).

```
0 1 0 0    dead  alive dead  dead
1 0 0 0    alive dead  dead  dead
0 0 1 1    dead  dead  alive alive
0 1 1 0    dead  alive alive dead
```

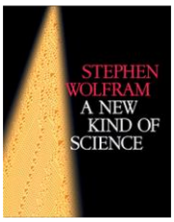
- Each cell has a ***neighborhood***. This can be defined in any number of ways, but it typically is a set of nearby cells. The most common neighborhoods are named after famous computer scientists who have done research on the subject, Von Neumann and Moore, whose namesake (*) is the most popular:

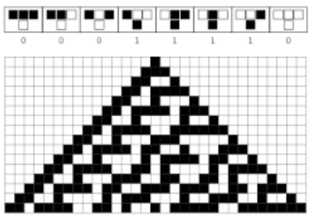
```
00000         00000            00000      00000
00000         00000            00000      00000
00000         00000            00000      00000
00000         00000            00000      00000
00000         00000            00000      00000

Von Neumann   Von Neumann extended   * Moore   Moore extended
```

The development of cellular automata systems is typically attributed to Stanisław Ulam (1909-1984) and John von Neumann (1903-1957), who were both researchers at the Los Alamos National Laboratory in New Mexico in the 1940s. Ulam was studying the growth of crystals and von Neumann was imagining a slightly bizarre world of self-replicating robots. Other pioneers in the field were Edward F. Moore (1925-2003) and George H.Mealy (1927-2010), who both worked at the legendary Bell Labs in the ‘50s.

Ulam, von Neumann, Moore, Mealy

The British mathematician Stephen Wolfram (b. 1959) dedicated almost 20 years of work to cellular automata, culminating in arguably the most authoritative textbook on the subject, the lengthy (1280 pages) *A New Kind of Science*[59] (available for free online, and a recommended reading), published in 2002. Wolfram discusses how cellular automata are not just neat tricks, but are relevant to the study of biology, chemistry, physics, and all sciences (including, just to name one, economics).





---

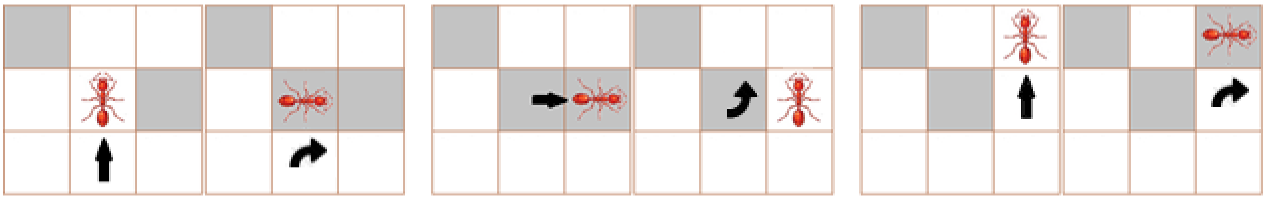
[58] Singular: *automaton*.

[59] This book is highly controversial: Wolfram’s conclusion that the universe is digital in nature and that its laws can be stated as programs have raised considerable debate.

Langton’s ant

One of the simplest, and most famous, cellular automata (which is also a Turing machine) is the brainchild of Chris Langton (b.1948), one of the founders of the studies on artificial life.

We can image a rectangular grid (**400x400** will be enough), made up of square tiles, either white or black. An ant moves across the grid, one tile at a time, either horizontally or vertically. When the ant steps on a white tile, the tile switches to black and the ant turns right; when it lands on a black tile, the tile switches to white and the ant turns left. The following images show the movement of the ant for a couple of steps:

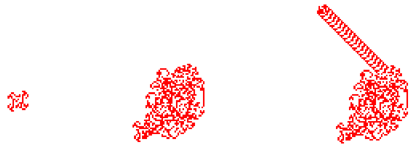


land on white tile | paint black, turn right | land on black tile | paint white, turn left | land on white tile | paint black, turn right

Normally, the playing field is reduced to a square matrix of points (one per tile), all initially white, and the ant is not drawn, so that only its traces remain on the screen, one pixel at a time. Of course, some points will be traversed several times, so they will frequently change color, but lots of colored points will be visible at any time. The outcome of the simulation is quite surprising, as the ant moves its first few hundreds of steps in symmetric patterns around the starting position, then moves in a somewhat chaotic way until about step 10000, before embarking in a repetitive pattern of exactly 104 steps, tracing what some call a *gun* and others a *highway* (see image).

The main issues of this simulation are the following:

- storing the value corresponding to the color of the tiles (the data structure is evidently a *matrix*);
- describing “turning” in terms of traversing the matrix (as usual, altering row and column indexes);
- choosing a calibration;
- deciding when the simulation should end.

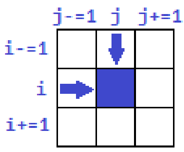


Step 400 | Step 10000 | Step 12000

1 Declaring the matrix is easy enough:

```
points = [[0 for x in range(400)] for y in range(400)]
```

The simplest choice is using `0` for white, `1` for black (more complex sets of values can be used for more advanced version of the game), which can be done at the beginning of the program.



2 Changing the indexes of the matrix is quite simple, as can be seen in the image to the right, where the assumption is that the blue cell is indexed by `[i][j]`. The real problem is defining “turning right” and “turning left” in *subsequent* steps. For example, the very first step of the ant is normally “right”, corresponding to `j+=1`; but one more `j+=1` would mean “straight”, and the correct move should instead be `i+=1`. On the other hand, `j+=1` would correctly mean “right” after going “up”. The conclusion is that actions depends on the previous situation, called the *state of the system*: this means that the ant *has memory* (both of the tiles and of its own movement). As it always happens, this complication means one more variable, storing the current direction of the ant.

3 The calibration should reflect the “integer” nature of the problem: one unit per pixel, without scaling factors, is the most common solution. For simplicity, a variable called `field` has been used, so as to avoid repeating the same constants all over the place (the value is `399`, not `400`, because matrices are zero-based):

```
field = 400
g.dimensions(field,field,0,field-1,field-1,0)
points = [[0 for x in range(field)] for y in range(field)]
```

4 As for ending the game, besides the usual interactive methods, it may be possible to limit the number of steps of the ant (from user input).

Track of a multi-colored ant

EXTENSION This simulation can be extended by choosing different rules (for example, with multiple colors, or more ways to move the ant, even in `3D`).

Conway's *Game of Life*

One of the classic problems in the trade is Conway's *Game of Life*[60], also with two cell states, aptly named dead (**0**) and alive (**1**). They may represent, for example, bacteria in a Petri dish, or rabbits in a field; the simulation may study how they survive, or maybe how they can be kept in check.

```
0 1 0 0 1   dead  alive dead  dead  alive
1 0 0 0 1   alive dead  dead  dead  alive
0 0 1 1 0   dead  dead  alive alive dead
0 1 0 0 1   dead  alive dead  dead  alive
0 1 1 0 0   dead  alive alive dead  dead
```

NOTE In theory, the matrix is infinite: but, as is always the case with computer models, we will have to set limits.

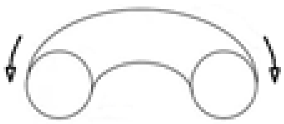
Initial cell states can be given, or chosen at random. The evolution of each cell **i,j** depends on its neighbors, of which there are eight (a Moore neighborhood, located by offsetting the matrix indexes from **-1** to **+1**):

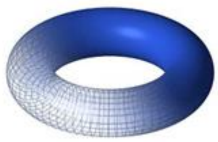
Row	Col	Row	Col	Row	Col
i-1	j-1	i+1	j	i+1	j+1
i	j-1	i	j	i	j+1
i+1	j-1	i+1	j	i+1	j+1

The game rules, as set by Conway[61], are four, and mimic what happens in real life with bacterial cultures (and possibly being the reason for christening the model as *cellular*):

- Any live cell with fewer than two live neighbors dies (from loneliness);
- Any live cell with two or three live neighbors lives on to the next generation;
- Any live cell with more than three live neighbors dies (from overcrowding);
- Any dead cell with exactly three live neighbors becomes a live cell (by reproduction).

One solution to the problem of finite space in a cellular automaton is the *toroidal world*, which consists in rolling the cell grid so that it has no vertical boundaries, becoming a cylindrical section: this is obtained by joining the first column with the last one (as when rolling up a piece of paper to build a cylindrical section). The resulting cylinder can be rolled again, joining the two remaining borders, as when building a loop from a cylinder (similarly to what plumbers do when they join pipes, only closing the circuit). The result is what geometers call a *toroid* (or, more jokingly, a *donut*).





With a matrix, the toroid can be easily obtained using the sentinels, copying the first row/column to the right/bottom of the last one, and the last row/column to the left/top of the first one (the four **corner cells** must be copied to the **opposite sentinels**). This technique has the additional advantage to avoid complications when addressing the neighbors of the boundary cells.

```
0 0 0 0 0 0 0        0 0 1 1 0 0 0
0 0 1 0 0 1 0        1 0 1 0 0 1 0
0 1 0 0 0 1 0        1 1 0 0 0 1 1
0 0 0 1 1 0 0        0 0 0 1 1 0 0
0 0 1 0 0 1 0        1 0 1 0 0 1 0
0 0 1 1 0 0 0        0 0 1 1 0 0 0
0 0 0 0 0 0 0        1 0 1 0 0 1 0
```

Empty sentinels   Sentinels in use

[60] John Horton Conway is a British mathematician (b.1937), active in number theory and in combinatorial game theory.

[61] Of course, anybody could set their own rules, but these are the originals ones, found throughout the literature.

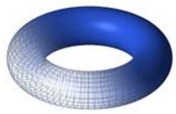
Curious *LIFE* patterns

The accuracy of our computer simulation can be tested with several particular configurations. Some patterns are still (they remain constant), some pulsate (changing orientation, with various periods), some just translate across the playing field (again, with various periods).

Still	Block	0000 0110 0110 0000
	Beehive	0110 1001 0110
	Loaf	0110 1001 0101 0010
	Boat	011 101 010
Pulsating	Blinker (period 2)	111
	Toad (period 2)	0111 1110
	Beacon (period 2)	1100 1100 0011 0011
	Pulsar (period 3)	0011100011100 0000000000000 1000010100001 1000010100001 1000010100001 0011100011100 0000000000000 0011100011100 1000010100001 1000010100001 1000010100001 0000000000000 0011100011100
Translating	Glider	01 001 111
	Lightweight spaceship (LWSS)	01111 10001 00001 10010
	Middleweight spaceship (MWSS)	000100 010001 100000 100001 111110
	Heavyweight spaceship (HWSS)	0001100 0100001 1000000 1000001 1111110

One extension of the LIFE game

From the vault, here is a beauty devised by a Canadian biological mathematician, Alexander Keewatin Dewdney, who wrote it for the *Computer Recreations* column on Scientific American (he succeeded none other than the great Martin Gardner), circa 1984. He later gathered many of his columns in the book *The Armchair Universe: an exploration of Computer Worlds*.



Dewdney's world is toroidal, just as in Life. It is a planet called Wa-tor because it is entirely covered with water. The only two occupants of this watery world are sharks and fish: the sharks eat the fish, and the fish live on a never-ending supply of plankton.

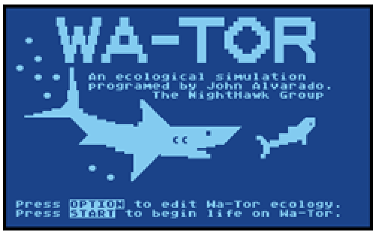
Time flows in discrete units, the ubiquitous chronons[62]: at each chronon, sharks and fish can move and several events may happen (sharks eating fish, spawning, death, …). The problem is to obtain stable populations, as most biologists dream: actually, the study of predators and prey relies on simulated ecologies like this one, which should be self-sustaining (or *autopoietic*, in the trade).

There are some considerations, taken from biology[63], which should be included in the simulations:

- Sharks are far fewer in number than fish: a rough estimate of nutrition efficiency is about **10%**, which also holds for agricultural production;
- Sharks and fish reproduce under specific conditions;
- Fish rely on their superior breeding skills; but, of the many newborns, few make it to adulthood;
- Sharks rely on their speed and hunting skills to locate and eat the fish;
- Hunger is an issue for sharks: not feeding in a certain interval leads to death by starvation;
- After a certain age, both fish and sharks die of natural causes;
- Extinction is also to consider: the sharks should not eat *all* the fish, lest they become extinct as well.

These rules are not too distant from Conway's; of course, there are additions. (new species, time intervals to define, hunting, and other conditions), but the basic concept can still be reduced to a matrix, populated by "numbers" which, in this game, can move, eat, spawn, and age.

Since the publication of Dewdney's column, there have been many implementations of the Wa-tor simulation, mostly in academic courses, but also as videogames and screen savers.



Some hints on how to write the computer simulations:

- The matrix is quite similar to Conway's, but it must be populated with two different kinds of animals: one simple solution is using positive or negative numbers to differentiate;
- The state variables of the system should include: age for reproduction, number of offspring, age for death, starving interval, plus any possible random modifications;
- In Dewdney's original formulation, events happen randomly: it is possible to regulate them differently;
- Hunting requires that sharks be able to examine the surroundings, in search for fish;
- Sharks could eat dead sharks, finding a further food source;
- There could be other rules for rotting carcasses;
- Gender (and related physical differences) could be introduced.

It is easy to see that many other things can be taken into account. The limit is the fantasy of the programmer.

---

[62] Dewdney could have been the first to use the term in computer simulations.

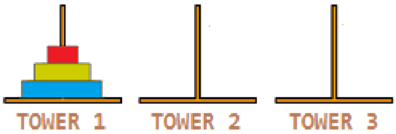
[63] There is a rich literature on this topic, dating back as far as the 19th century, with the independent work of Alfred Lotka (USA, 1880-1949) and Vito Volterra (Italy, 1860-1940), which led to the so-called *Lotka-Volterra predator-prey model*. Later, the Canadian mathematician Crawford Stanley "Buzz" Holling (b.1930) worked extensively on this model, adding several contributions.

The towers of Hanoi (cellular solution)

This is a classical mathematical puzzle, invented by the French Édouard Lucas (1842-1891), a specialist in number theory. According to one legend, quite probably fabricated by Lucas himself[64], a Brahmin temple in Kashi Wishwanat (India, or, more exactly, Bharat) contains three posts holding a total of **64** golden disks, all with different diameters, originally forming a pyramid on the first post (or a conical shape, if you wish). Priests of the temple spend their time shifting the disks from one post to the other, trying to move the pyramid of disks to the third post, following two simple rules:

- Only one disk can be moved at a time;
- A larger disk cannot be placed onto a smaller one.

The puzzle is quite common in a children's version, with a limited number of wooden or plastic disks:



NOTE: The game requires the third post as a target: we use a looser rule, where any target post is considered legit.

According to the legend, when the tower is reconstructed on the third post, the world will end. No need to worry, though: with **64** disks, even moving one per second, it would take more than **585** billion years to complete the task. The simple reason is the power of exponentiation: with **n** disks, the minimum number of moves to solve the puzzle is 2^n-1.

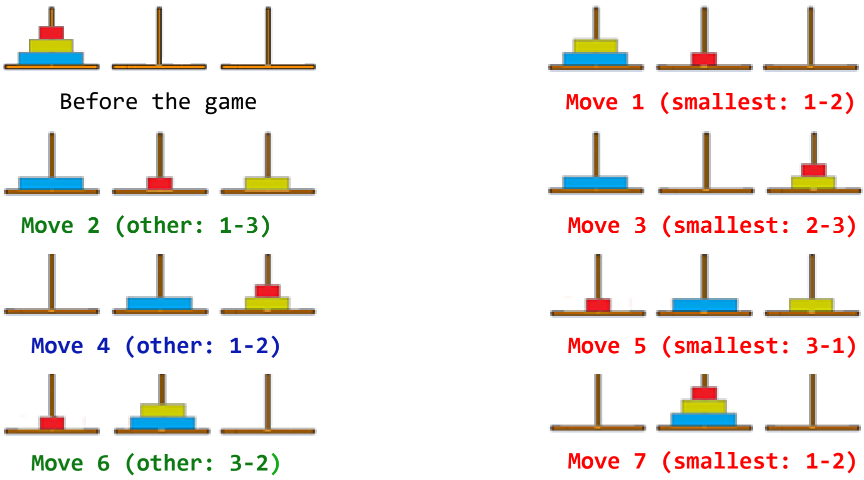
Without analyzing too deeply the mathematical reasoning, we will use a very simple algorithm, which relies on the repetition of two steps:

1. Move the smallest disk (pictured in **red**) to the right;
2. Make any *other* move (that is, the only move that leaves the smallest disk in place).

Both moves require a quick explanation:

1. "to the right" from tower **2** wraps around to tower **0** (or from tower **3** to **1**, in a **1**-based matrix);
2. "any other move", leaving the smallest disk alone, leaves no choice, since the disks are all different.

Clearly, in computer programming, this is a description of a loop. Since we know that the number of moves is odd, we will cheat a little, and make the first move before the loop, which will contain the steps in reverse order (step 2, then step 1), and thus will finish smoothly, avoiding a further filter[65].



We can notice that the smallest disk (**red**) is moved the majority of times (about half, or 2^{n-1}), the next disk (**green**) half that number (2^{n-2}), down to the largest disk (**blue**), which is moved only once (2^0). Recalling some notions of binary arithmetic, this process is associated to *repunits* (sequences of digits all equal to **1**).

Following the strict rule of allowing only the third post as target would require reversing the direction of movements when the number of disks is odd. We can avoid this subtlety.

---

[64] What makes the legend suspect is the fact that Lucas marketed and sold the game under the pseudonym of *N. Claus de Siam*, which is an anagram of *Lucas d'Amiens* (Amiens was his birthplace). Also, depending on the storyteller, sometimes the temple is located in other places (including Hanoi, Vietnam), and sometimes it is instead a monastery.

[65] After all, it is not really cheating, as the first and last disk to be moved is *necessarily* the smallest one.

In the world of computer simulations, there are several solutions to this puzzle, some quite elegant, and some with fancy graphics and animations; in our case, we will use a cruder "cellular" method, in line with the spirit of this chapter.

First of all, the `n` disks can be represented by numbers, stacked in a matrix, ranging from `1` (smallest) to `n` (largest).This way, the initial and final configurations of the system (reversing the towers) could be:

```
row 0      5 0 0      0 0 5        0 5 0
row 1      5 0 0      0 0 5        0 5 0
row 2      4 0 0      0 0 4   OR   0 4 0
row 3      3 0 0      0 0 3        0 3 0
row 4      2 0 0      0 0 2        0 2 0
row 5      1 0 0      0 0 1        0 1 0
```

> **NOTE**
> filling the matrix with any given height `n` is a quite elementary task.

The blue numbers, stored in three sentinels in row `0`, can be used to represent the *height* of each tower. Using column `0` for actual content, instead of sentinels, can simplify computing the movements of the disks.

The action of moving disks can be represented by *swapping* values of the matrix, with the sentinels as props. For example, once determined the source and destination posts, the first move should consist of the following ideal steps:

```
5 1 0      5 1 0      5 1 0      4 1 0
5 0 0      5 0 0      5 1 0      5 1 0
4 0 0      4 0 0      4 0 0      4 0 0
3 0 0      3 0 0      3 0 0      3 0 0
2 0 0      2 0 0      2 0 0      2 0 0
1 0 0      1 0 0      0 0 0      0 0 0
```

step 1	step 2	step 3	step 4
Add `1` to height of destination post	Consider the numbers indicated by the sentinels	Swap said numbers	Subtract `1` from height of source post

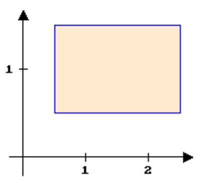
It is necessary to track the position of the smallest disk with a supplementary variable. When its value is known, and considering the two remaining posts, it becomes trivial determining the next move, which only needs to verify two conditions:

- One of the posts is empty: this becomes the destination post, while the other is chosen as the source;
- Neither posts is empty: the source post is the one with the smaller number at the top.

The disk at the top of tower (column) `p` can be located with `a[p,a[0,p]]`. Clearly, these considerations must be done before moving anything (that is, before what is listed as ***step 1*** in the preceding example).

The rectangles representing the disks (as viewed from the side) can be declared and drawn with the designer object, where a `rectangle` object has four properties `l`, `t`, `w`, and `h` (the coordinates of the top-left corner and the dimensions).

An example of how to use a rectangle is the following:




```
r = Rectangle()
r.l = 0.5
r.t = 1.5
r.w = 2
r.h = 1
r.draw(g)                      #    default: black border, white interior[66]
r.draw(g,BLUE,BLANCHEDALMOND)  #    setting border, interior
```

Rectangles can also be used to draw the posts: clearly, the disks should be drawn after the posts, to give a better visual effect.

Of course, as is always the case, a good project should carefully plan the positions and the dimensions of the elements of the simulation.

A further improvement of the exercise could be the animation of the moving disks, which requires several new considerations, including the height to which raise the moving disk and its physical destination.

---

[66] Or whatever combination has been set for the designer object.

Turing machines

or:

a working simulator

Cellular automata Combinatorics

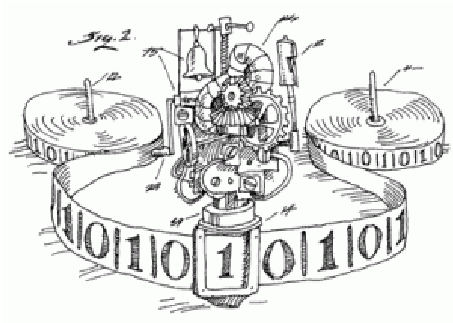
What, if anything, is a Turing machine?

Alan Mathison Turing (1912-1954) was a brilliant British mathematician, considered one of the founding fathers of computer science. Among other things, he was one of the codebreakers who cracked the German ciphers during the Second World War, working at a secret location in Bletchley Park, not far from London[67]. He was also a world-class distance runner, who barely missed qualifying for the 1948 Olympic marathon (coming in fifth at the UK trials).

Turing is credited with giving the soon-to-be-born computer science a sound theoretical basis, with his hypothetical device called *Turing Machine*, described in 1936. A Turing machine is a virtual device, capable of reproducing any algorithmic process: it is not a working technology, as it exists only in principle, but it describes an apparatus totally equivalent to the programmable computer[68].

This is how Turing himself describes his namesake in a 1948 essay:

...an infinite memory capacity obtained in the form of an infinite tape marked out into squares, on each of which a symbol could be printed. At any moment there is one symbol in the machine; it is called the scanned symbol. The machine can alter the scanned symbol and its behavior is in part determined by that symbol, but the symbols on the tape elsewhere do not affect the behavior of the machine. However, the tape can be moved back and forth through the machine, this being one of the elementary operations of the machine. Any symbol on the tape may therefore eventually have an innings[69].



A Turing machine “operates” on a tape

A Turing machine comprises the following components:

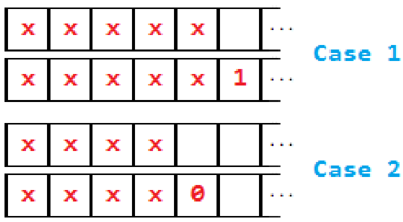
1. A **tape** (also called **ribbon**) subdivided into adjacent cells, each of which can contain a symbol belonging to a given, finite alphabet;
2. A **read/write head**, which can scan any cell of the tape: this element is an *automaton* (pl. *automata*; a fancy name for a *finite state machine*), and is the working part of the contraption;
3. A **transition table** for the automaton, describing its action; depending on the state it is in and the symbol it encounters, the automaton can:

   - Change or maintain the symbol on the tape;
   - Move the tape in either direction;
   - Change or maintain its state

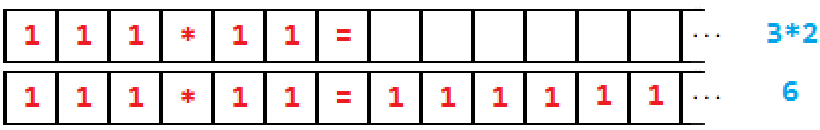
In Turing’s design, it is the tape that is moved: most computer simulations, however, rely on moving the read/write head, which results in simpler programs. Also, a working model requires a finite tape.

But the most important thing about the tape is that it is the only *memory* of the machine: anything that needs to be stored will occupy one or more cells (be they intermediate or final results).

For example, let us consider a quite simple Turing machine which computes the parity of a sequence of symbols: the result (either `0` or `1`) will be written in the first empty cell following the sequence. In this case, the alphabet has only three symbols (`x`, `0`, and `1`), where `0` and `1` are reserved for the result, while the initial sequence should only contain `x`’s. in this case, the machine only needs to traverse the ribbon once.



A more complicated machine computes a product between integer numbers, stored in unary notation (where a number `n` is represented by `n` symbols). The alphabet contains the symbol `1` and the operation signs (`*` and `=`). In this case, the ribbon will be traversed back and forth a number of times. The example shows how the ribbon should contain the two factors at the beginning, and the complete result at the end:



[67] For more on this little-known chapter of the war, see `https://www.bletchleypark.org.uk/content/hist`

[68] This is even more remarkable considering that the first electronic computers were a decade into the future.

[69] The expression “to have an innings” means being dedicated some working time.

The parity machine

Our Turing machines start at the left end of the ribbon (which is finite, contrary to the Turing model), in the lowest possible state (normally **0** or **1**). The example shows the initial and final states of the tape, including the position of the read/write head:

`Initial ribbon`	x	x	x	x	x		
`R/W head position`	*						
`Final ribbon`	x	x	x	x	x	1	
`R/W head position`						*	

The parity of a sequence of changes with each symbol encountered: so this machine only requires two states, switching between them at each cell. It is not necessary to actually *count* the symbols, which would be quite a task for a Turing machine (it is possible, but not worth the effort). For example, the machine could start in state **0**, then change to state **1** after the first cell, then back to state **0** after the second, and so on. When the machine encounters an empty cell, it will store there the result, then stop. Of course, the result will be a function of the current state: if it is **0**, the sequence is even, so the machine will print a **0**; the only other possibility is state **1**, in which case the machine will print **1**.

This is how this machine works:

Action	RIBBON							Movement
Change to state 1	x	x	x	x	x			Move Right
	*							
Change to state 0	x	x	x	x	x			Move Right
		*						
Change to state 1	x	x	x	x	x			Move Right
			*					
Change to state 0	x	x	x	x	x			Move Right
				*				
Change to state 1	x	x	x	x	x			Move Right
					*			
End (blank) encountered	x	x	x	x	x			
						*		
Write result	x	x	x	x	x	0		STOP
						*		

The transition table can be summarized thus:

State 0: either change to state "1" or print result				
0	X		R	1
0		0	H	
State 1: either change to state "0" or print result				
1	X		R	0
1		1	H	

The table can be stored in a text file (shown in the box), where the first line contains a sample initial sequence (so as to avoid having to type one each time the program is run).

Each line contains 5 entries, corresponding to the working of the machine:

- Current state
- Symbol scanned
- New symbol
- Movement (L=left; R=right; H=Halt; E=Error)
- New state

```
111111
0x R1
0 0H
1x R0
1 1H
```

A working model

The text file containing the transition table can be fed to a **PHP** page, which can simulate the corresponding machine. It can be written with any text editor, with the only requirement of a **.txt** extension. The upper part of the page contains the dynamic ribbon.

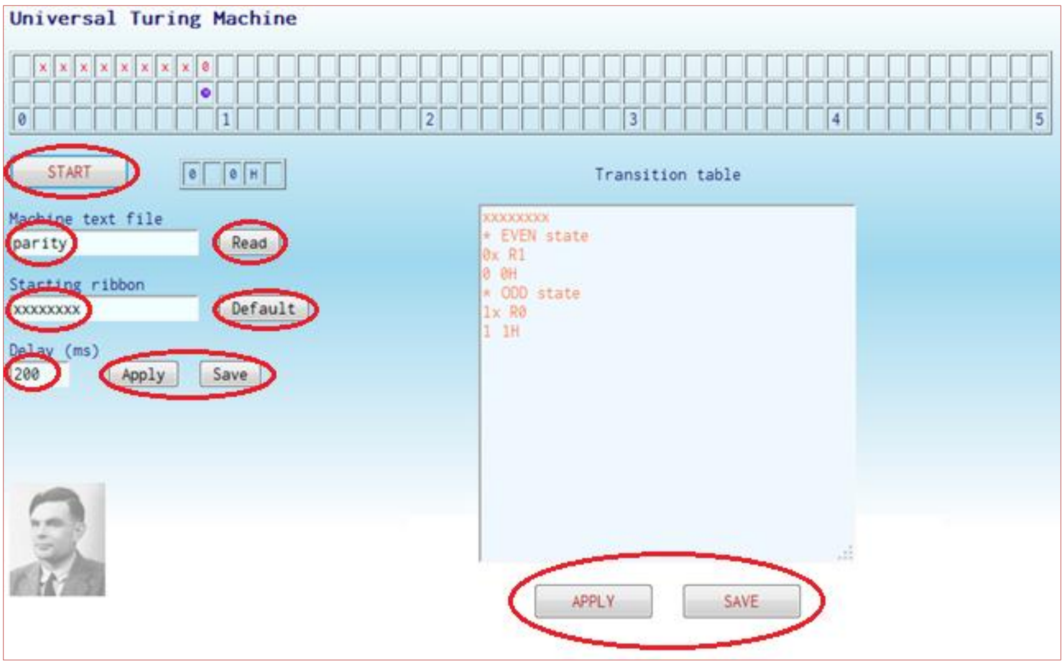
The necessity for a **PHP** page stems from the fact that there is interaction with the file system. Due to security reasons, a local server, known from previous courses, should be used (most public servers deny uploading files unless doing so from portals).

- After keying in the file name, clicking on the **read** button will load the content of the text file into the *textarea* to the right (labeled "Transition table").
- Clicking on **START** will activate the machine, using the first line of the text file as the content of the ribbon;
- The **START** button morphs to **STOP**, which can be used to halt the machine;
- It is possible to change the initial sequence in the appropriate textbox (or return to the sequence stored in the text file by clicking on the **default** button);
- It is possible to change the delay between steps, so as to speed up or slow down the machine, either temporarily (**apply**) or permanently (**save**);
- The transition table con be edited and saved directly from the page, without the need of a text editor;
- The **APPLY** button activates the modified transition table without saving it (not strictly recommended, since the page could abort, thus losing all pending changes).

Some rules about the transition table:

- The machine starts with the lowest state (normally **0** or **1**, but there are no restrictions);
- The read/write head starts always from position **1**;
- The commands (**L**, **R**, **H**, and **E**) can be written either in *lowercase* or in *UPPERCASE*;
- The **H** and **E** commands can be either in the third or fourth position of their command line;
- States are identified by a single characters: state **9** can be followed by state **A**, **B**, **C**, and so on;
- The default delay is **300** milliseconds;
- In the unlikely case of deleting a symbol (not a recommended practice), an underscore **_** should be the "new symbol" (the third entry in the transition table);
- A wildcard character (**?**) can be used as a shorthand for "any symbol scanned" (second entry of the transition table), excluding those explicitly declared in other rows;
- Comments can be added, starting with an asterisk (*****).

The following is a snapshot of the Web page:



Sorting a string made up by two symbols (A and B)

Initial ribbon	A	A	B	B	A	B	A			
Final ribbon	A	A	A	A	B	B	B			

This problem is much simpler than it may appear: it is solvable with just 3 states (plus one to clean up the ribbon): the trick is, whenever a B is found, travel to the right in search of an A, then swapping the two cells. Each symbol B encountered must be marked, so as to memorize that it is being processed: since the only memory of the machine is the tape, an intermediate symbol will be stored in its place (in this case, a). Of course, when no As are found after a B, the solution is complete; in this case, it is possible that one cell may still contain an a, so the tape should be traversed again, to clean up the undesired symbol with a proper B.

This is how this machine works:

Finding a B	A	A	B	B	A	B	A			Right
Marking the B with a	A	A	a	B	A	B	A			
Finding an A	A	A	a	B	A	B	A			Right
Changing the A into B	A	A	a	B	B	B	A			
Finding the a	A	A	A	B	B	B	A			Left
Changing the a into A	A	A	A	B	B	B	A			
Next pair	A	A	A	B	B	B	A			
After swapping	A	A	A	A	B	B	B			
No A's following this B	A	A	A	A	a	B	B			Right
Ribbon after cleanup	A	A	A	A	B	B	B			Left, then STOP

The transition table can be summarized thus:

State 1: find a "B", then mark it with "a"				
1	A		R	1
1	B	A	R	2
1			H	
State 2: find an "A" to the right, then change it into a "B"				
2	B		R	2
2				
2	A	B	L	3
State 3: store the "A" where the "B" (now "a") was				
3	B		L	3
3	a	A	R	1
State 9: clean up any suspended "a" from the ribbon (notice the wildcard ?)				
9	?		L	9
9	a	B	L	9
9			H	

Finally, this is the content of the text file:

```
AABBABA
1A R1
1BAR2
1 H
2B R2
2
2ABL3
3B L3
3aAR1
9aBL9
9A L9
9B L9
9  H
```

Multiplying two numbers in unary notation

Initial ribbon (3*2)	1	1	1	*	1	1	=							
Final ribbon (6)	1	1	1	*	1	1	=	1	1	1	1	1	1	

For each digit of the first factor, the second factor (which is after the * sign) must be duplicated to the right of the = symbol. To avoid ambiguities, the digits of the first factor are marked with X, those of the second with 2, and those of the result with 3. A cleanup state A restores all the digits; a safety cleanup state B is used when the second factor is 0 (as in 111*=, which would leave the ribbon as X11*=).

State 1: find a digit in the first factor				
1	*		H	
1	1	X	R	2
States 2-3: find the second factor (after *)				
2	1		R	2
2	X		R	2
2	=		L	B
2	*		R	3
3	=		L	B
3	1	2	R	4
States 4-6: duplicate the second factor after =				
4	?		R	4
4		3	L	5
5	?		L	5
5	1	2	R	4
5	*		R	6
6	?		R	6
6	=		L	8
State 8: restore the second factor and find another digit of the first factor				
8	2	1	L	8
8	?		L	8
8	1	X	R	2
8			R	A
State A/B: clean up				
A	?	1	R	A
A	*		R	A
A	=		R	A
A			H	
B	?		L	B
B	X	1	H	

This is the content of the text file:

```
111*11=  4? R4   821L8   A?1RA
1* H     4 3L5   8? L8   A* RA
11XR2    5? L5   81XR2   A= RA
21 R2    512R4   8  RA   A H
2X R2    5* R6           B? LB
2= LA    6? R6           BX1H
2* R3    6= L8
3= LA
312R4
```

COMBINATORICS

or:

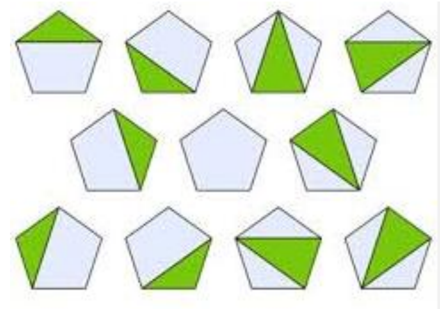
How to count without counting

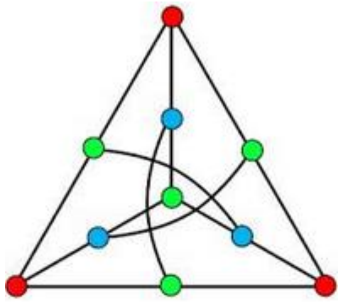
Turing machines Fractals

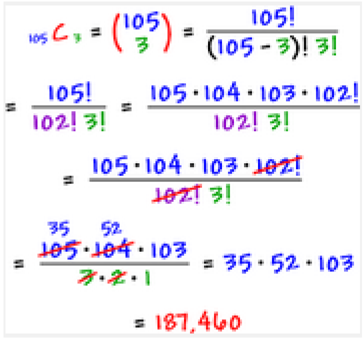
What, if anything, is *combinatorics*?

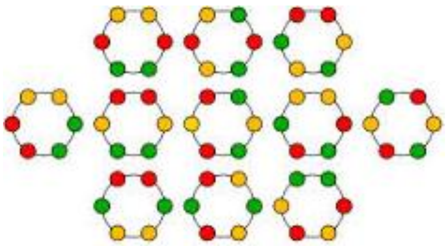
Combinatorics, sometimes called *combinatorial mathematics*, is the branch of mathematics that deals with countable, discrete structures. One subset of such studies which is frequently encountered in computer simulations regards *finite sets*, where elements can be enumerated and chosen in several ways.

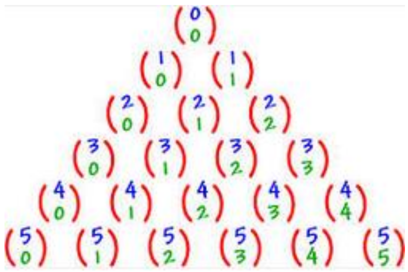
Elementary algebra, geometry, probability, statistics, graph theory, group theory, number theory (the Queen of Mathematics, no less), to name just a few, all rely heavily on finite sets: for this reason, combinatorics has many implications in mathematics.





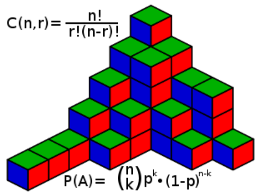






In addition, there are countless technological applications of combinatorics which make our life easier (though not always, one might add), for example:

- cellphone communications (error correcting codes)
- game programming
- website organization
- airline booking systems
- construction projects.



A note on notations

It sometimes happens in mathematics that different nations, and different mathematicians, use different notations to refer to the same thing. This is even more evident in combinatorics. Two examples which will be seen in this chapter are the following:

$$C'(n,k) = CC(n,k) = \left(\!\binom{n}{k}\!\right) \qquad\qquad D_{n,k} = P_{n,k} = {}_nP_k$$

The art of arranging things

Permutations

Simply put, *permutations* are ordered arrangements of elements in distinguishable sequences (also called *lists*). If the cardinality of a given set is n, each permutation comprises all n elements, and is characterized by the order in which the elements are listed.

For example, suppose a gentleman owns four hats. The question is: in how many ways can he store them on the hat rack?

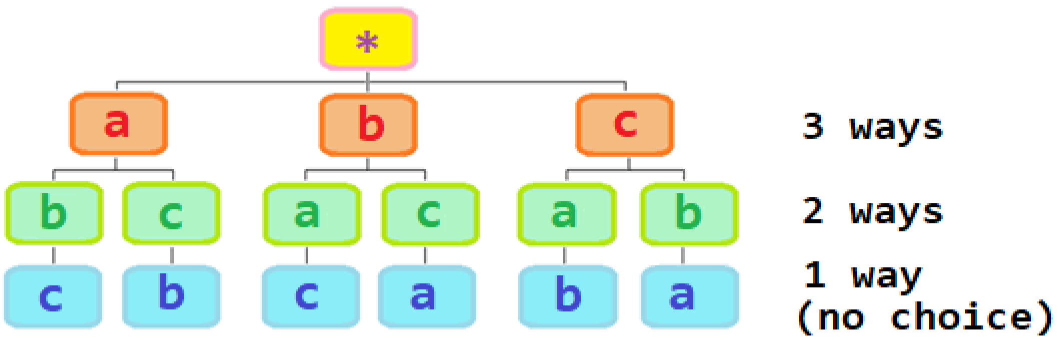
Let's define the set as {t,f,b,g}; with some patience, we can show that there are 24 arrangements:

(t,f,b,g) (t,f,g,b) (t,b,f,g) (t,b,g,f) (t,g,f,b) (t,g,b,f)
(f,t,b,g) (f,t,g,b) (f,b,t,g) (f,b,g,t) (f,g,t,b) (f,g,b,t)
(b,t,f,g) (b,t,g,f) (b,f,t,g) (b,f,g,t) (b,g,t,f) (b,g,f,t)
(g,t,f,b) (g,t,b,f) (g,f,t,b) (g,f,b,t) (g,b,t,f) (g,b,f,t)

There is a simple formula to evaluate how many permutations are possible in a given set.

✓ Of course, the only element in a singleton, such as {a}, can be chosen in just 1 way.
✓ In a set containing 2 elements, such as {a,b}, there are 2 ordered couples: (a,b) and (b,a).
✓ In the general case, in a set containing n elements the reasoning can be extended:
  - The first element of the list can be chosen in n distinct ways;
  - The second element of the list can be chosen in n-1 distinct ways;
  - The third element of the list can be chosen in n-2 distinct ways;
  - And so on…
  - Once n-1 elements have been chosen, there is only 1 way to choose the remaining one.

Sometimes the procedure is depicted with diagrams such as the following (with three elements):



Since each choice is independent from all the others, the total number of permutations for n elements is obtained by multiplying the possible choices (this is the *multiplication principle*). Notations vary from country to country, but the general formula remains:

$$_nP_n = P_n = n*(n-1)*(n-2)*...*2*1 = \prod_{i=1}^{n} i$$

The product of all positive integers between 1 and n is called the **factorial** of n, and is denoted with the symbol n!. The notation was introduced by the French mathematician Christian Kramp (1760-1826), in the treatise *Elements d'arithmétique universelle*, written in 1808. Some with a weird sense of humor call the factorial "n shriek" or "n bang".

4!=24 corresponds to the empirical solution listed above for the arrangements of hats on the rack. With three hats, the solution would then be 3!=6.

The factorial of 0 is defined as 0!=1. There seems to be little logic in this, but it is useful to maintain an algebraic balance in certain calculations (more on which later).

Computing factorials, and a bit of recursion

The values of the factorial function increase in an extremely fast way. Consider the first few values:

```
1 2 6 24 120 720 5040 40320 362880 3628800 39916800 479001600 6227020800
```

This sequence soon exhausts (at `12!`) the value limits of integer variables. Python does not distinguish between numeric types, so it can compute any factorial up to `170!`, which, at `307` digits[70], is the maximum value for the processing capabilities of most programming languages (still, with the limit of `15` significant digits).

The following is an elementary code for such a function:

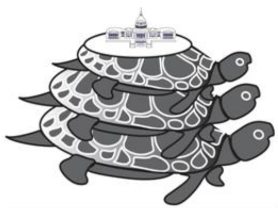
```
def factorial(n)
    f = 1
    for i in range(2,n+1):
        f*= i
    return f
```

The factorial function is a typical example of a *recursive* procedure (something that can be described in terms of itself). Analyzing the computation, it can be seen that the factorial of any number `n` is `n` times the factorial of `n-1`:, which itself is `n-1` times the factorial of `n-2`, and so on, all the way down to `0`, rather like a set of Russian dolls:

```
5! = 5 * 4 * 3 * 2 * 1 = 5 * 4!
  4! = 4 * 3 * 2 * 1 = 4 * 3!
    3! = 3 * 2 * 1 = 3 * 2!
      2! = 2 * 1 = 2 * 1!
          1! = 1 * 0!
            0! = 1
```

The reasoning fits the following definition of factorial, rather symbolic:

```
1. 0! = 1
2. n! = n*(n-1)!
```



“It’s turtles all the way down!”

From this last definition, the solution comes almost naturally:

```
def factorial(n):
    if n<0:            (**)
        return None
    if n==0:           (*)
        return 1
    else:
        return n * factorial(n-1)
```

The factorial is calculated in a cascading way, always in terms of smaller numbers: of course, there must be a starting (or ending, depending on perspective) point, or our program would generate an infinite sequence of calls, just like in the turtle story[71]. This requires a condition, marked with `(*)`, which, not surprisingly, reflects both the recursive definition and the curious definition of `0!`. The condition marked with `(**)`, offers a way out when some incautious program provides a negative argument.

It should also be noted that recursive functions, simple as they are to state, are quite costly on resources: any function call places new copies of the function code, and a new set of variables, on the stack (the memory space available for data). With many calls, this could soon clog up the memory, causing the program to stop. So it is wise to use recursion carefully, and only if strictly necessary: an iterative solution may be less elegant, but it is almost always faster and more reliable. Limitations or not, in the academic world recursion is widely studied and highly appreciated.

---

[70] About `7.25741561530799E+306`. `171!` exceeds the highest floating-point value, `1.7976931348623157e+308`.

[71] There are several versions of this story. More or less, it goes that an astronomer (whose actual name varies, depending on provenance) is confronted, after a lecture, by a little old lady who says: “What you have told us is rubbish. The world is really a flat plate supported on the back of a giant tortoise”; when the scientist asks: “What is the tortoise standing on?”, the final answer of the lady is: “Good question, young man: but it's *turtles all the way down*!”

Dispositions

In some situations, not all the elements of the set are included in the list, but only a given number `k` (with `k<=n`), originating arrangements called *dispositions* (sometimes, *k-dispositions*). As with permutations, in dispositions order does matter.

For example, the question could be how many *different* 3-digit numbers can be obtained from a set of 5 digits (for those who prefer hats, it is like counting in how many ways 3 hats can be chosen out of 5). It is important to stress out that the digits chosen should all be different (that is, without *repetition*). Arrangements such as `212` or `225` are not considered (at least, not yet: more on this soon). The considerations are quite similar to those on permutations:

- The first element of the list can be chosen in `n` distinct ways;
- The second element of the list can be chosen in `n-1` distinct ways;
- And so on…
- Once `k-1` elements have been chosen, there are `n-(k-1)` ways to choose the remaining one.

$$D_{n,k} = P_{n,k} = {}_nP_k = n*(n-1)*(n-2)*...*(n-k+1) = n^{\underline{k}}$$

The symbol $n^{\underline{k}}$ (where the exponent $\underline{k}$ is underlined) is called a *falling factorial*.

Also, multiplying and dividing the formula by `(n-k)!`, we can obtain a formula with pure factorials:

$${}_nP_k = n*(n-1)*(n-2)*...*(n-k+1) * \frac{(n-k)!}{(n-k)!}$$

$${}_nP_k = \frac{n!}{(n-k)!}$$

The answer to the sample question, then, is `5!/2!` (`60`), as shown with the digits `{1,2,3,4,5}`:

```
Starting with 1   123 124 125 132 134 135 142 143 145 152 153 154
Starting with 2   213 214 215 231 234 235 241 243 245 251 253 254
Starting with 3   312 314 315 321 324 325 341 342 345 351 352 354
Starting with 4   412 413 415 421 423 425 431 432 435 451 452 453
Starting with 5   512 513 514 521 523 524 531 532 534 541 542 543
```

To further test the rule, let us consider the extreme cases, `k=1` and `k=n`: the former should evaluate to `n`, as there are `n` ways to pick one object from a collection of `n`, and the latter should be equivalent to the number of permutations. We can easily see that the results hold, with another proof that `0!` is needed:

$${}_nP_1 = \frac{n!}{(n-1)!} = n \qquad {}_nP_n = \frac{n!}{(n-n)!} = \frac{n!}{0!} = n!$$

Actual computations follow the same dictum as regular factorials: an iterative solution, though less elegant, is faster than a recursive one.

Iterative solution	Number of operations

`def dispositions(n,k):`
`    f = 1`
`    for i in range(n-k+1,n+1):`
`        f*= i`
`    return f`
	`k multiplications`

Factorial solution	

`def dispositions(n,k):`
`    return factorial(n) // factorial(n-k)`
	`n multiplications (1st factorial)`
	`(n-k) multiplications (2nd factorial)`
	`1 division (quotient)`
	`Total: 2*n-k+1`

For example, ${}_{10}P_4$ requires `4` multiplications with the iterative solution and `16` with the factorial one, plus one division, which on most platforms is generally slower than a multiplication. With these numbers, this is hardly a problem: things can be a lot different, though, when millions of calculations are needed.

Combinations

In certain situations, order does not matter, as with Lotto or Bingo. In this case, we define a *combination* as any subset of a given set, where the elements are all different (and with no ordering). Given the cardinality **k** of the subset (with **k<=n**), it is also called a *k-combination*.

The number of combinations (which, of course, is smaller than the number of dispositions) is obtained by dividing the number of dispositions by the number of equivalent choices (which is **k!**).

$$C(n,k) = C_{n,k} = D_{n,k} \;/\; k! = n^{\underline{k}} \;/\; k! = \frac{n!}{k!\,(n-k)!} = \binom{n}{k} \qquad \text{(n choose k):}$$

Combinations spring up in many unexpected situations. One which is known from elementary algebra is the binomial coefficient, thus called because they appear in the expansion of the power of a binomial:

$$\textbf{General formula:}\ (a+b)^n = \sum_{k=0}^{n} \binom{n}{k} a^k b^{n-k}$$

$$\textbf{Example:}\ (a+b)^5 = a^5 + 5a^4b + 10a^3b^2 + 10a^2b^3 + 5ab^4 + b^5$$

Binomial coefficients can be found with Pascal's triangle[72], which algebra textbooks usually represent in a pyramidal shape, whereas, in computer science, it assumes a matrix form. Any number of the triangle can be obtained from the preceding row, summing the neighboring elements (in matrix form, the one to the left and the one immediately above). In the following example, the red **4** has been obtained by summing the elements **3** and **1** just above it (the third column will be dealt with later):

1	1 0 0 0 0 ...	row 0 (sum=1)
1 1	1 1 0 0 0 ...	row 1 (sum=2)
1 2 1	1 2 1 0 0 ...	row 2 (sum=4)
1 3 3 1	1 3 3 1 0 ...	row 3 (sum=8)
1 4 6 4 1	1 4 6 4 1 ...	row 4 (sum=16)
1 5 10 10 5 1	1 5 10 10 5 1	...
...	...	row n (sum=2^n)

In general, the element in the k^{th} position of the n^{th} row (starting from **0**) is $\binom{n}{k}$.

Sometimes, calculations can be shortened by using this formula, as the **example** shows:

$$\binom{n}{k} = \binom{n}{n-k} \qquad \binom{100}{97} = \binom{100}{3}$$

Another useful properties of binomial coefficients is the following:

$$\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}$$

Poker in the movies

In some movies involving poker, it seems like just a matter of time before the hero is dealt a *royal flush* (ace, king, queen, jack, and 10 of the same suit), or a *4oak* (four of a kind). The truth is a little bit different. Let us calculate the number of possible hands (with a full deck of **52** cards, used in most American versions of the game, including *draw poker*):

$$C(52,5) = \binom{52}{5} = \binom{52}{47} = \frac{52*51*50*49*48}{5*4*3*2*1} = \frac{52*51*50*49*48}{5*4*3*2*1} = 2598960$$

Since there are only **4** royal flushes (one per suit), the probability of being dealt one is

4/2598960=0.0000015

To put this number into perspective, consider that a player who were dealt **20** hands of poker every night of the year could expect one royal flush in about **89** *years*. Considering only straight flushes, there are **40** of them, so everything must be adjusted by a factor of **10** (probability **0.000015**, about **9** years waiting).

What about a 4oak? There are **13** of them, but we must take into account the fifth card, which can be any of the remaining **48**, giving us **13*48=624** possibilities. Thus, a 4oak is **156** times more likely to happen than a royal flush: the chances of being dealt one are **624/2598960=0.00024**. The same player of the preceding example should wait only about seven months.

[72] After the French mathematician Blaise Pascal (1623-1662). In Italy, it is called Tartaglia's triangle, after Niccolò Fontana (1500?-1557), of Brescia, called Tartaglia (=the stammerer) because of a speech impediment.

A little bit of poker – counting the hands

These calculations are based on the standard American deck of **52** cards, where the lowest card rank is **2**. In many European countries, including Italy, the lowest card rank is given by **11** minus the number of players (who are, optimally, four or five, yielding **7** or **6**): in these variants, a *flush* will beat a *full house*.

Royal flush

This is easy: only one **A-K-Q-J-10** combination is possible for each suit, so the total is **4**.

Straight flush

The lowest possible straight is **5-4-3-2-A**, so there are **10** possible highest-ranking cards (from **A** down to **5**) for each suit: the total is **40** (4*10). The number of non-royal flushes is **36**.

Four of a kind/4oak

There are **13** ranks of this combination; the remaining card can be chosen in **48** possible ways, so the number of possible hands is:

$$13 * 48 = 624$$

Full house/full boat

This hand has 3oak and a pair. Now we can forsake counting and use combinatorics, with a tricky formula:

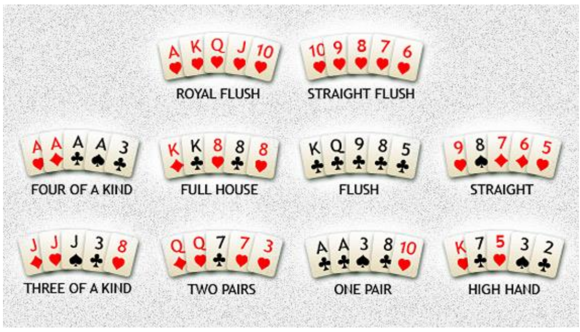
$$\binom{13}{1} * \binom{4}{3} * \binom{12}{1} * \binom{4}{2} = 3744$$

A little bit tricky, but not too much. The factors are: number of ranks for the 3oak – ways of choosing the three cards – number of ranks for the pair – ways of choosing the pair.

Flush

We have five cards for any suit (this explains the 4^1 factor), but we must discard the straight flushes, royal or not:

$$\binom{13}{5} * 4^1 - 40 = 5108$$



Straight

As seen before, there are **10** straight ranks. Each card can be of any suit, so there is a 4^5 factor to be included, and, again, the royal flushes must be discarded. So the calculation is:

$$10 * 4^5 - 40 = 10200$$

Three of a kind/3oak

Contrary to the full house, the two remaining cards *must* have different ranks from the 3oak, with a 4^2 factor to consider all possible suits:

$$\binom{13}{1} * \binom{4}{3} * \binom{12}{2} * 4^2 = 54912$$

Two pairs

This, and the next one, are tricky:

$$\binom{13}{2} * \binom{4}{2} * \binom{4}{2} * \binom{11}{1} * 4^1 = 123552$$

The factors are: ranks of the pairs – ways of choosing each pair (two equal factors) – ways of choosing the remaining card – number of suits for the remaining card.

Pair

$$\binom{13}{1} * \binom{4}{2} * \binom{12}{3} * 4^3 = 1098240$$

The factors are: rank of the pair – ways of choosing the pair – ways of choosing the remaining cards – number of suits for each of the three cards.

High card/nothing

This can be calculated by subtracting all the other combinations from the total of possible hands:

$$2598960 - (40 + 624 + 3744 + 5108 + 10200 + 54912 + 123552 + 1098240) = 1302540$$

The numbers show how the vast majority of combinations are *pairs* and *high cards*.

Arrangements with repeated elements

Permutations with repetition

Many combinatorial problems (hats, dinner guests, batting orders) involve sets with distinct (that is, *different*) elements. In some cases, though, elements can be repeated, so the same value could appear more than once. This happens, typically, with letters and digits.

For example, let us consider the set of letters: **{s,l,e,e,p}**. The words **sleep** and **sleep** are identical, regardless of which **e** is used: it is quite intuitive to see that total number of arrangement is half of what would result without repetition (since there are two interchangeable letters, giving place to **2!=2** possible permutations between themselves). So, out of the **120** arrangements of **5** letters, only **60** count.

The reasoning can be extended for any subset of identical elements, where each reduction is independent from the others (which calls for the multiplication of cases). Supposing that one element is repeated **a** times, another **b** times, another **c** times, and so on, the general formula becomes:

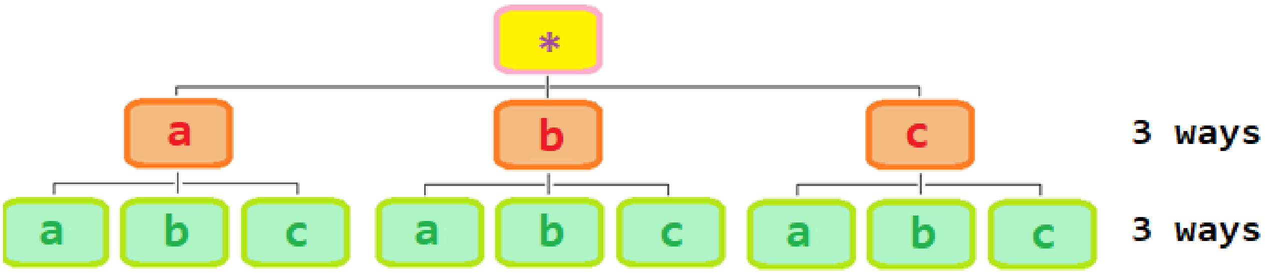
$$P'(n,a,b,c) = {}_nP_{abc\ldots} = P_n^{abc\ldots} = \frac{n!}{a!b!c!\ldots}$$

In another example, let us consider the set **{a,b,b,b}**. We can see that the permutation **abbb** has six equivalent arrangements (**abbb**, **abbb**, **abbb**, **abbb**, **abbb**, **abbb**), in which the order of the repeated elements does not really matter. The same goes for each of the remaining permutations (**babb**, **bbab**, and **bbba**). This means that the total number of permutations should be divided by six (the number of equivalent arrangements for each permutation, or the number of permutations of the repeated element).

So, as expected, P_4^3 evaluates to $P_4 \ / \ P_3 = \frac{4!}{3!} = \frac{24}{6} = 4$.

Dispositions with repetition

The general case for dispositions is when the number of elements to be chosen is not limited by **n**, and repetitions are allowed. This is akin to picking one bingo number, then putting it back into the bin, so that it might be extracted again (this becomes necessary when **k>n**, or we would run out of elements). The following is an example with two extractions:



The reasoning behind evaluating the number of dispositions is similar to that on permutations, but not quite:

- The first element of the list can be chosen in **n** distinct ways;
- The second element of the list can be chosen in **n** distinct ways;
- And so on…(up to any desired or required limit, generally called **k**)

So the number of k-dispositions with repetition boils down to **n** choices multiplied a number of times corresponding to the number **k** of extractions; the general formula, then, has no restrictions (only **n>=0** and **k>=1**), and no reduction in the factors to be counted:

$$D'(n,k) = D'^{\,k}_{n} = n^k$$

Combinations with repetition

As with permutations, combinations can be considered with repeated elements (called *k-multicombination*). In this case, the notation is as follows:

$$C'(n,k) = CC(n,k) = \left(\!\!\binom{n}{k}\!\!\right) = \binom{n+k-1}{k} \qquad \text{(n multichoose k)}$$

Miscellaneous stuff

Sitting guests at a dinner table

One typical class of problems regards how to sit a certain number of guests at a table, or: “In how many ways can n guests sit at a table?”. In the simplest cases, there are no preferences: any guest could pick any place at random. More complex problems could add further constraints (for example: “one gentleman and one lady”, or “gentlemen on one side, ladies on the other”, or “best man and bridesmaid close to the newlyweds”, and so on).

If the dinner took place at a dais, the actual places would matter, and the answer would be, as expected, $n!$, or, calling the guests by number:

123 132 213 231 312 321

If, on the other hand, the table were round, there would be no preferences nor priorities, and only the relative order would matter (being either to the left or to the right of other guests). In this case, some choices would be equivalent: for example, with **123**, **231**, and **312**, guest **2** would be to the right of guest **1**, and guest **3** to the right of guest **2**. In this case, the sitting choice of the first guest should not be counted, and the answer would then be $(n-1)!$, with the following two cases (the other ones are just rotations around the table):

123 132

Choosing your socks blindly

In other instances, one is required to choose matching socks from a closet in a dark room (for some reason, combinatorial mathematicians often get dressed up in the dark). One classic problem is as follows: suppose you had **4** blue socks, **8** red, and **6** green. What is the probability of randomly choosing a matching pair?

Number of possible pairs: $C(18,2) = \binom{18}{2} = 153$

Number of possible **blue** pairs: $C(4,2) = \binom{4}{2} = 6$

Number of possible **red** pairs: $C(8,2) = \binom{8}{2} = 28$

Number of possible **green** pairs: $C(6,2) = \binom{6}{2} = 15$

> **BONUS**
> What is the *minimum* number of socks to pick if you wanted to get two matching ones?

So, the chances to pick two matching socks are $(6+28+15)/153 = 0.32$, or about one in three.

Choosing your donuts

You are at the local candy store. If you had a choice of 5 types of donuts, in how many ways could you choose 3 of them? In this case, the availability can be considered unlimited, so the solution lies in a k-multicombination:

$$C'(5,3) = CC(5,3) = \left(\!\binom{5}{3}\!\right) = \binom{5+3-1}{3} = \binom{7}{3} = 35$$

The problem should not be confused with a similar one, where you only have those 5 donuts, in which case the choice would be a k-disposition:

$$\binom{5}{3} = 10$$

> **BONUS**
> solve the case where the choice is *up to* three donuts

Surprise fact

The following formula shows how the sum of the coefficients of any row of Pascal’s triangle is a power of **2**:

$$\sum_{k=0}^{n} \binom{n}{k} = 2^n$$

But there’s more. As an interesting twist, let us consider the subsets of $\{1,2,3\}$ (of which there are eight):

$\emptyset$ - $\{1\}$ - $\{2\}$ - $\{3\}$ - $\{1,2\}$ - $\{1,3\}$ - $\{2,3\}$ - $\{1,2,3\}$.

It turns out, perhaps with some surprise (but not quite), that, for any set of cardinality n, its subsets number 2^n. For this reason, the set of all subsets of a given set A is called the *power set* of A, and symbolized with 2^A.

Additional problems

1

Out of a standard American deck of 52 cards:

a) How many poker hands of five cards can be dealt?
b) How many hands contain 2 aces?
c) How many hands contain *at least* 2 jacks?
d) How many hands contain 3 cards with consecutive ranks? (e.g.: **J-10-9**, **4-3-2**, **A-K-Q**)
e) Extend the preceding problem by excluding all the combinations with 4 consecutive ranks (e.g.: with **Q-J-10** and neither kings nor nines), which could lead to a straight.
f) Extend the preceding problems by excluding also the cards with the same rank as the original three, so that none of them can produce a pair.
g) Extend the preceding problems by excluding that also the two remaining cards form a pair, thus reducing the hand to a "high card".

2

How many ways can the letters of the following words be rearranged to form new words (not necessarily with sense)?

a) ANYHOW
b) ANA
c) HAWTHORN
d) ANNA
e) HANNAH
f) SENESCENCE

3

a) How many ways can you arrange 16 people into 4 rows of 4 desks each?
b) How many ways can you choose 4 groups of 4 distinct people from 16 overall?
c) How many ways can you pair up 8 boys and 8 girls?

4

We have a group of 6 men and 8 women, eligible for a committee comprising 3 men and 3 women.

a) How many committees can be chosen?
b) How many if two men are incompatible?
c) How many if two women are incompatible?
d) How many if one man is incompatible with one woman?

$$\left(\!\binom{5}{2}\!\right) = \binom{5+2-1}{2}$$
$$= \binom{6}{2} = \frac{6!}{2!(6-2)!} = \frac{6!}{2!4!}$$
$$= \frac{6 \cdot 5 \cdot \cancel{4!}}{2 \cdot 1 \cdot \cancel{4!}} = \frac{30}{2} = 15$$

5

How many ways can you distribute 10 candies among 7 kids, provided that each kid gets at least one?

6

A group of 10 tennis players organize a doubles tournament.

a) How many sets of couples can be chosen?
b) How many matches can be played among all the couples?

7

Bonus: how many matches are needed to play a 64-player tennis tournament?

8

This is the nightmare that still haunts yours truly. Prove that: $\binom{n}{k} = \binom{n}{n-k}$

Solutions

1

a) This is a simple combination of 52 objects, taken 5 at a time.

$$C(52,5) = \binom{52}{5} = 2598960$$

b) In the deck there are 4 aces (A) and 48 non-aces (N). What we are looking for is something like:

AANNN

This corresponds to 2 aces out of 4 and 3 cards out of the remaining 48. The solution is:

$$C(4,2) * C(48,3) = \binom{4}{2} * \binom{48}{3} = 17296$$

c) This problem is an extension of the preceding one, but with more possibilities, as in:

JJNNN + JJJNN + JJJJN

The solution is then:

$$C(4,2)*C(48,3) + C(4,3)*C(48,2) + C(4,4)*C(48,1) = 103776 + 4152 + 48 = 108336$$

Another solution is subtracting NNNNN + JNNNN from the number of total hands.

$$2598960 - (C(48,5) + C(4,1)*C(48,4)) = 2598960 – (1712304+778320) = 108336$$

d) There are 12 possibilities of having three consecutively ranked cards (from A-K-Q down to 3-2-A), without restrictions regarding suits; the remaining two cards can be chosen among the 49 left:

$$12 * 4^3 * \binom{49}{2} = 768 * 1176 = 903168$$

e) If, for example, we had a Q-J-10 hand, the remaining two cards could be neither kings nor nines, so the 49 cards are reduced to 41:

$$768 * \binom{41}{2} = 768 * 820 = 629760$$

We might also consider two special cases: with A-K-Q a 2 is not relevant, as is a king with 3-2-A. In those cases, the allowed cards are 45 instead of 41:

$$768 * \binom{45}{2} = 768 * 990 = 760320$$

f) In this case, the cards with the same rank of the original three are 9, so the allowed cards can be 40 (problem **d**) or 32 (problem **e**, with 36 for the special cases):

$768*\binom{40}{2}=768*780=599040$	$768*\binom{32}{2}=768*780=380928$	$768*\binom{36}{2}=768*630=483840$

g) In each case, there is a certain number of ranks allowed: 10, 8, and 9, respectively. The possible pairs, and the corresponding combinations to be subtracted from the preceding totals, can then be summarized thus:

$\binom{10}{1} * \binom{4}{2} = 10*6 = 60$	$\binom{8}{1} * \binom{4}{2} = 8*6 = 48$	$\binom{9}{1} * \binom{4}{2} = 9*6 = 54$
599040 – 60 = 598980	380928 – 48 = 380880	483840 - 54 = 483786

"Four jacks!"

(*The Sting*, 1973)

The Sting features one of the best poker scenes ever seen in a movie. Yet, as far as probability goes, it is quite implausible.

In one occasion, three players (Jameson, Lombard, and Shaw) call a two pair, a 3oak, and a flush *in the same hand*. The probabilities for each of these combinations are, respectively: 0.048, 0.021, and 0.002; the aggregate probability is the product of the three, a tiny 0.000002.

The final hand is even worse: Lonnegan and Shaw square off with two 4oak, which combine for a probability of 0.0000000576. Of course, the hand is rigged, but that's another story (also, Shaw is actually Gondorff).

Normally, in fact, poker hands are more like a pair beating a high card, or something even more boring. The real thrill of the game is in the stakes.

2

These are simple problems of permutations with repetition, with the general formula summarized as:

$$P_n^{abc\ldots} = \frac{n!}{a!\ b!\ c!\ \ldots}$$

a) ANYHOW $\quad P_6^{0} = \frac{6!}{0!} = \frac{720}{1} = 720$

b) ANA $\quad P_3^{2} = \frac{3!}{2!} = \frac{6}{2} = 3$

c) HAWTHORN $\quad P_8^{2} = \frac{8!}{2!} = \frac{40320}{2} = 20160$

d) ANNA $\quad P_4^{2,2} = \frac{4!}{2!\ 2!} = \frac{24}{4} = 6$

e) HANNAH $\quad P_6^{2,2,2} = \frac{6!}{2!\ 2!\ 2!} = \frac{720}{8} = 90$

f) SENESCENCE $\quad P_{10}^{2,4,2,2,2} = \frac{10!}{2!\ 4!\ 2!\ 2!\ 2!} = \frac{3628800}{192} = 18900$

3

a) In this problem, the desks and rows are only decoys. It doesn't matter how the desks are arranged. You could number them **1** through **16** and the problem becomes a simple permutation of **16** objects, where order counts. The answer is a staggering **16! = 20,922,789,888,000**.

b) This is a problem of combinations, choosing **4** objects out of **16**, then **4** more out of the remaining **12**, and so on. The solution is:

$$C(16,4)*C(12,4)*C(8,4)*C(4,4) = \binom{16}{4} * \binom{12}{4} * \binom{8}{4} * \binom{4}{4} = 63{,}063{,}000$$

The last factor evaluates to **1**: this is understandable, since there is only one way of choosing the last four people, once **12** have been selected.

c) It sounds like there are **2** sets of **2** people to consider permuting, but really we are only permuting only one set of **8**. Think of the boys as in a fixed order: **B1 B2 B3 ... B8**. Then the girls can be arranged in **8!** ways. Each arrangement corresponds to one pairing with the boys: first girl in the arrangement with **B1**, second girl in the arrangement with **B2**, etc.

4

a) Choose three out of six and three out of eight (simple combination formulas), then multiply:

$$C(6,3) * C(8,3) = 20 * 56 = 1120$$

b) The **2** incompatible men reduce **20** to **16**, because they can be involved in $\binom{4}{1} = 4$ combinations with any other man:

$$16 * 56 = 896$$

c) The **2** incompatible women reduce **56** to **50**, because they can be involved in $\binom{6}{1} = 6$ combinations with any other woman:

$$20 * 50 = 1000$$

d) The **2** incompatible subjects subtract **C(5,2)*C(7,2)** from the total, *not* from the partials:

$$1120 - 210 = 810$$

5

First, simplify the problem by giving the kids one candy each; this can be done in only one way, since all the candies are equal. This way, we get rid of **7** candies, at the same time satisfying the required condition of "at least one candy each".

Then reformulate the problem as "distribute **3** candies among **7** kids". This is equivalent to choosing **7** elements in **3** ways, with repetition (any kid could get more than one candy, or no candies at all):

$$CC(7,3) = \left(\!\binom{7}{3}\!\right) = \binom{7+3-1}{3} = \binom{9}{3} = 84$$

By tweaking the formulas we can obtain:

$$\binom{7+3-1}{3} = \binom{3+7-1}{3} = \binom{3+(10-3)-1}{3} = \binom{10-1}{3}$$

Applying to the general case of **m** candies and **n** kids, it is easy to verify the equality:

$$\left(\!\binom{n}{m-n}\!\right) = \binom{n+(m-n)-1}{m-n} = \binom{m-1}{m-n}$$

6

These are simple combination problems (ten players, two at a time; five couples, two at a time):

a) $C(10,2) = \binom{10}{2} = 45$

b) $C(5,2) = \binom{5}{2} = 10$

7

The first round takes **32** games, the second round **16**, and so on, until we reach the final (**1** game):

$$32 + 16 + 8 + 4 + 2 + 1 = 63$$

Another simpler reasoning is the following: in each match, **1** player is eliminated; to eliminate **63**, then, that's exactly the number of matches to play. After all, *there can be only one*[73].

If this reminds the reader of the binary number system, that's because it should: all the rounds can be led to the powers of **2**.

8

$$\binom{n}{k} = \frac{n!}{k!\,(n-k)!} = \frac{n!}{(n-n+k)!\,\,(n-k)!} = \frac{n!}{(n-k)!\,\,(n-(n-k))!} = \binom{n}{n-k} \quad \textbf{QED}$$

---

[73] *Highlander* (1986).

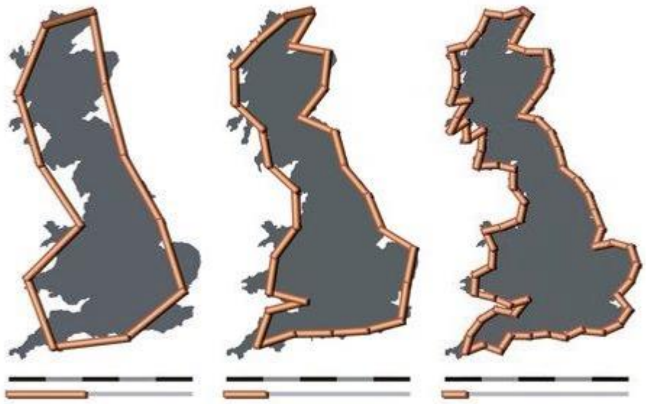
A little bit of fractals

Combinatorics Appendices

What, if anything, are *fractals*?

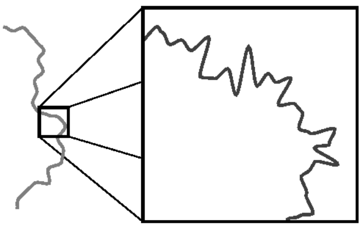
This story starts with a rather simple question, posed in 1967 by Benoit Mandelbrot (a Polish-French mathematician): "how long is the coast of Britain"? Geography books and atlases provide a simple answer, derived from the Ordnance Survey[74]: "*The coastline length around mainland Great Britain is **11,072.76 miles [17,819.88 km]***". But how accurate is that? It turns out that there is no simple answer, due to the so-called *coastline paradox*: the length of a stretch of coastline strongly depends on the scale of measurement. In fact, Mandelbrot's question was just an excuse to introduce the hitherto mysterious world of fractals[75],

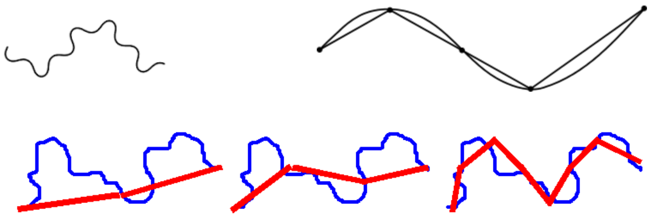
We soon discover that the shore of England (or any other shore, for that matter) is much more complex than geography would allow, breaking up into a convoluted shape with a seemingly infinite number of turns and bends. To begin with, it is easy to see how the polygon used to approximate the coastline strongly influences the measurement:



Benoit Mandelbrot
(1924-2010)

The more we zoom in on the image, the more difficult it becomes to choose the actual coastline, down to the single grains of sand. We will also notice that, whatever the zoom factor, the same shapes seem to appear again and again, leading to an infinite complexity with repeated geometric patterns: this is what Mandelbrot called *self-similarity*. There shapes are also nowhere differentiable: their very peculiarity makes it impossible to stop the repeating patterns and find a true tangent in any of their points.





The term *fractal* refers to the *fractional* dimensions, non-integer values that defy the common topological dimensions of geometric shapes (0 for points, 1 for lines, 2 for surfaces, and so on), being associated with the complexity of self-similar shapes.

The story of self-similarity and fractals traces its roots to such mathematical legends as Leibnitz, Weierstrass, Cantor, Klein, and Poincarè: of course, these shapes can best be appreciated with the help of computer graphics, and this is the main reason why they emerged from relative obscurity only in the 1980s.

Astonishingly, fractals are ubiquitous in the natural world (of course, without the benefit of infinity), both organic and inorganic, where growth patterns often lead to self-similar patterns:

Cauliflower

Snowflake

Nautilus

Landscape

---

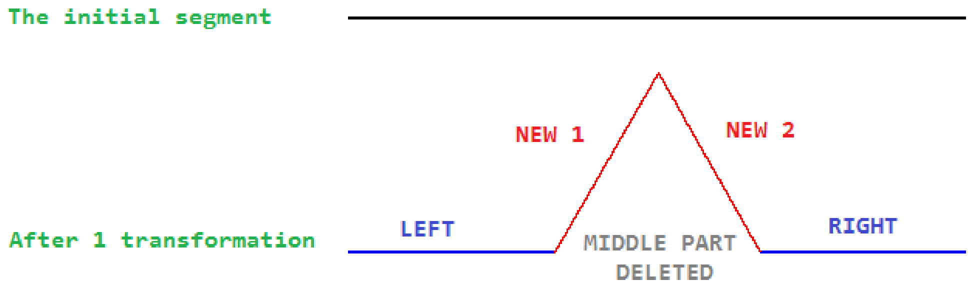
[74] The spelling *ordnance* is correct, referring to weaponry, not to legal rules, as is the case with *ordinance*. Incidentally, the name betrays the origin of this mapping agency to the military.

[75] Actually, Mandelbrot himself did not use the term *fractal* until 1975.

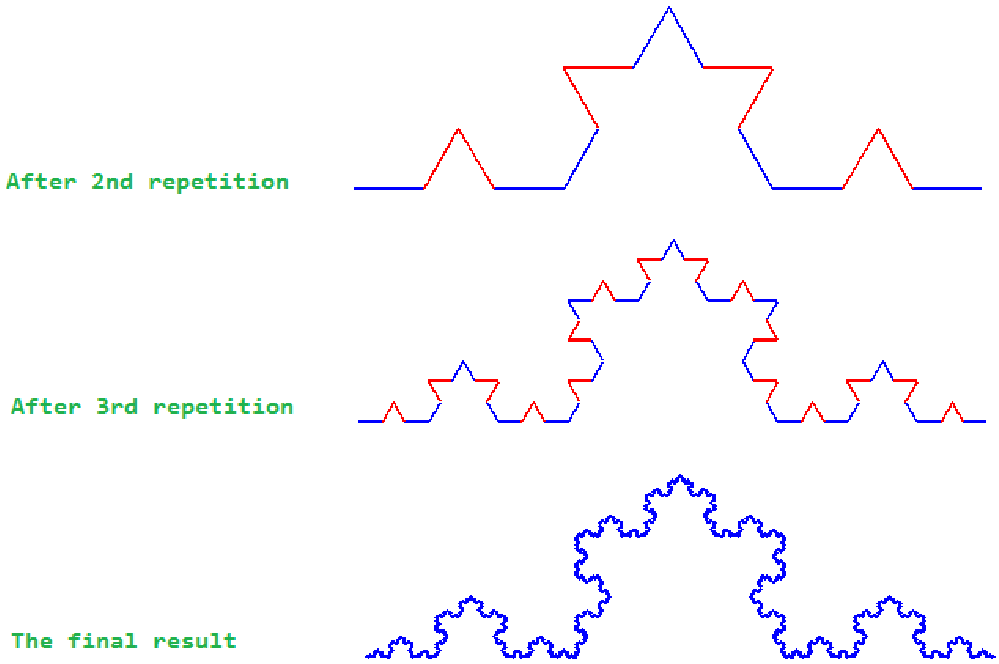
Koch's snowflake

This curve is a fractal studied by the Swedish mathematician Niels Fabian Helge von Koch (1870-1924). In true fractal tradition, this is a continuous curve, nowhere differentiable. In fact, his 1904 work was entitled "*Sur une courbe continue sans tangente, obtenue par une construction géométrique élémentaire*" ("*On a continuous curve without tangent, obtained through an elementary geometric constriction*").

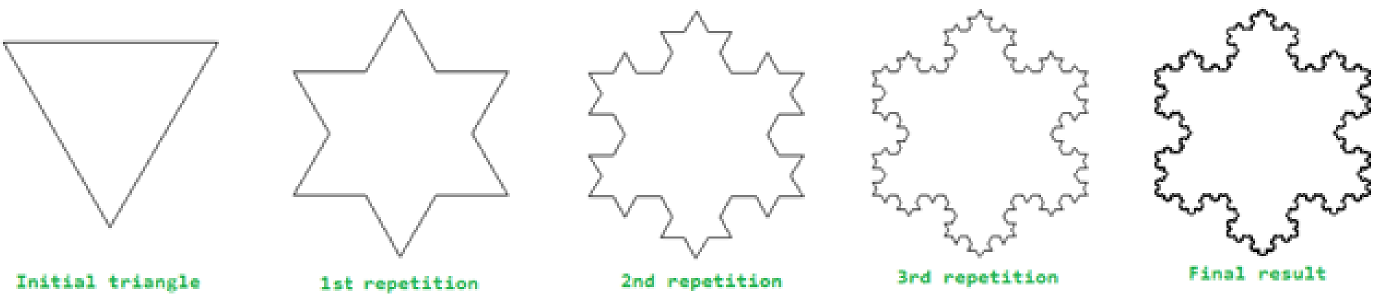
The curve can be constructed recursively, starting from one segment, dividing it into three (equal) parts, and replacing the middle one with two new segments, forming an equilateral triangle with the deleted segment.



The same process is then applied to all the resulting four segments, giving place to something resembling a spur or a caltrop. When repeating the procedure at will, the curve starts to lose any similarity with man-made implements, looking more and more like a cauliflower, one of the symbols of recursive shapes.



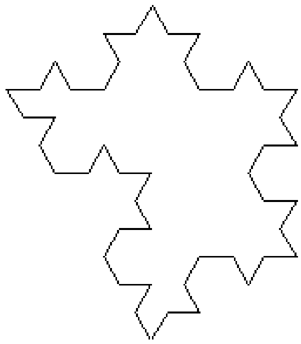
The actual "snowflake" is built starting from an equilateral triangle, applying the aforementioned process to all the sides, leading to a hexagonal pattern, from which result the reason for the moniker is evident:



The recursive function, called **flake**, operates the construction, then applies itself to each of the four resulting segments. Considering that each step triples the number of segments, the length of the segments at repetition **n** should evaluate to $LEN/3^n$ (where **LEN** is the length of the original segment), but there is some loss due to approximation, hence the condition for feasibility:

```
def flake(s,LEN,n):
    z = s.length()
    if z >= .99 * LEN / 3**n:          #   .99 due to approximation
        p1 = s.a.projection(s.b,z/3)
        p2 = s.b.projection(s.a,z/3)
        L = Dash(s.a,p1)
        R = Dash(p2,s.b)
        new1 = Dash(p1,p2)
        new1.rotate(pi/3)
        new2 = Dash(new1.b,p2)

        s.delete()                     #   Replace segment "s" with the
        g.draw(L,2)                    #   four segments of the construction
        g.draw(R,2)
        g.draw(new1,2)
        g.draw(new2,2)

        flake(L,LEN,n)                 #   Apply same construction to each
        flake(R,LEN,n)
        flake(new1,LEN,n)
        flake(new2,LEN,n)

def loop(LEN,n):
    i = 0
    g.start(1)
    while g.alive() and i<n:
        g.clear()
        flake(s,LEN,i)
        i+=1
        g.tick(1)

g = Designer(500,500,-2.5,2.5,2.5,-2.5)
s = Dash(-2,0,2,0)
s.draw(g,2)
g.foreground(BLUE)
LEN = s.length()
loop(LEN,8)
```

For the complete snowflake, we need to start with an equilateral triangle. It is necessary to orientate all the sides in the same way, or weird effects could take place, as in the following example, where one of the diagonal sides develops towards the interior:



The Mandelbrot set

The coiner of the word *fractal*, as well as the main popularizer of this branch of mathematics, was, of course, Benoit Mandelbrot, a truly international figure: he was born in Poland, raised in France and later transferred to the USA, where he worked in such places as the Institute for Advanced Studies in Princeton[76], IBM and Harvard.

During his time at IBM, Mandelbrot had an awful lot of computer time available, and that helped him both in his research and in the popularization of fractals, which he described as "beautiful, damn hard, increasingly useful". And while it is true that the first reaction of the scientific world was suspicious, fractals are now ubiquitously used, with applications such sciences as physics, astronomy, and medicine. Not surprisingly, one of the most natural links of fractals is with chaos theory.

Benoit Mandelbrot is forever linked with a particular set of complex numbers (the Mandelbrot set), named after him by a French colleague, the renowned joker Adrien Douady (1936-2006), who also made many contributions to the field of fractals, among which the *Douady rabbit*.

Simply put, the Mandelbrot set contains the complex numbers **c** such that the function:

$$f_c(z) = z^2 + c$$

does not diverge when iterated from **z=0**. The sequence can be stated recursively:

$$z_0 = 0$$
$$z_{n+1} = z_n^2 + c$$

Without analyzing the mathematical details, suffice it to say that any time the absolute value of **|z|** exceeds **2**, the sequence diverges. The main point is when to stop the sequence, that is, how many iterations must be done before accepting a point as satisfying the property. In fact, there is a wide variance, as the following examples show:

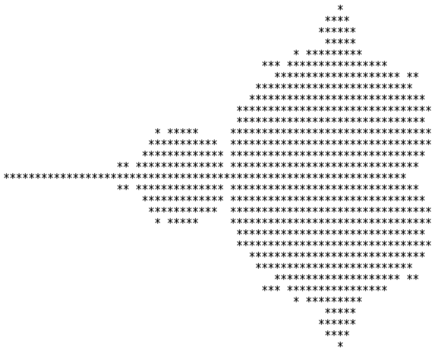
c	step	z	|z|
-1.5 + 0.4i	1	-1.5 + 0.4i	1.55
	2	0.59 - 0.8i	0.99
	3	-1.79 - 0.54i	1.87
	4	1.41 + 2.35i	2.74
-1.5 + 0.1i	1	-1.5 + 0.1i	1.50
	2	0.74 – 0.2i	0.77
	3	-0.99 - 0.2i	1.01
	4	-0.55 + 0.49i	0.74
	5	-1.43 - 0.44i	1.50
	6	0.36 + 1.36i	1.41
	7	-3.23 + 1.08i	3.41
0.2 + 1.9i	1	0.2 + 1.9i	1.91
	2	-3.37 – 2.66i	4.29

For a point to be included in the set, the starting values must be small (in fact, the set is contained in the circle with radius **2**, which is logical enough). The following is one such sequence:

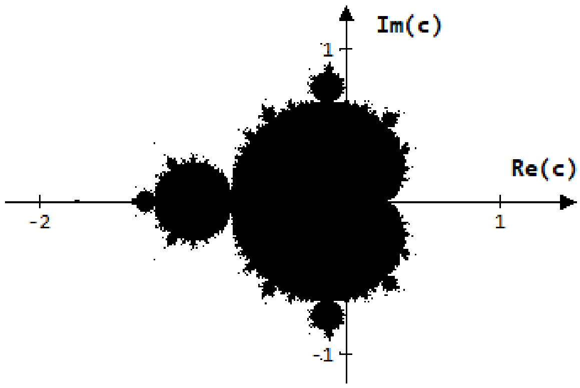
c	step	z	|z|
-1 + 0.2i	1	-1 + 0.2i	1.02
	2	-0.04 – 0.2i	0.20
	3	-1.04 + 0.22i	1.06
	4	0.03 – 0.25i	0.25
	5	-1.06 + 0.18i	1.08
	6	0.09 – 0.19i	0.21
	7	-1.02 + 0.17i	1.04
	...	...	...

[76] Among the guests of this institutions were Albert Einstein, John von Neumann, and Kurt Gödel.

Plotting the points of the Mandelbrot set in black on the complex plane, the result is a curious kidney shape:

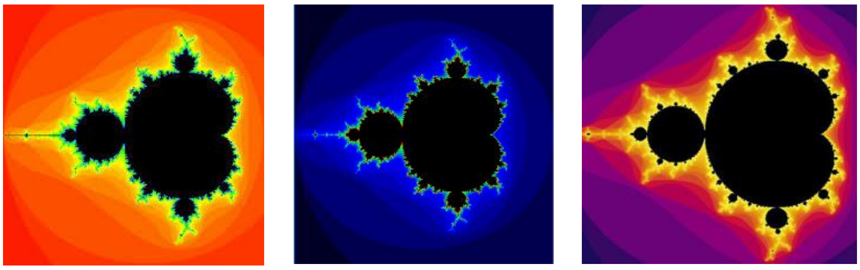


The first published picture of the Mandelbrot set used asterisks to approximate the points

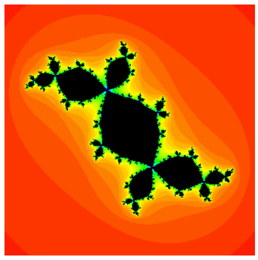


A more familiar B/W view

What is peculiar is that the most interesting points of the Mandelbrot set are not its own, but those *outside* it, particularly at the boundaries. In fact, while the Mandelbrot points are normally drawn in plain black, by associating different colors to the number of iterations needed to verify the divergence of the sequence, striking images can be drawn, such as the following (where a homage to Douady is also included):



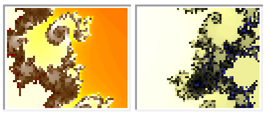
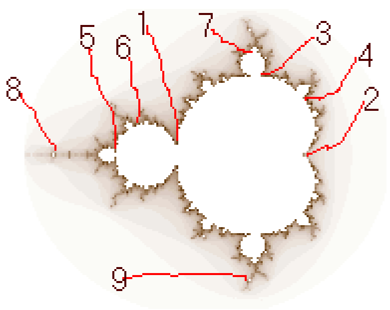
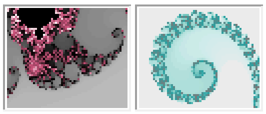
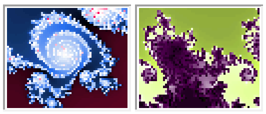
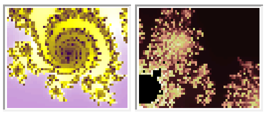
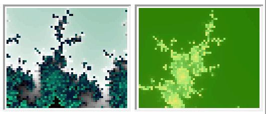
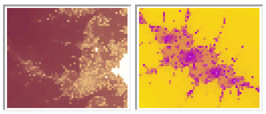
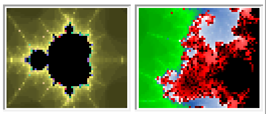
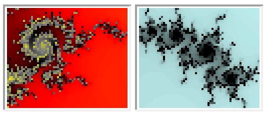
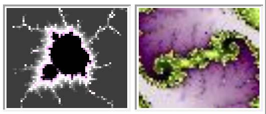
Different renditions of the Mandelbrot set



The Douady rabbit

The Mandelbrot set features an incredible number of different patterns, and its different zones have been named according to their geometric characteristics. Among the scores of Web sites dedicated to this set, this "map" is adapted from:

http://www.nahee.com/Derbyshire/manguide.html

1. **Seahorse Valley** *-0.75, 0.1*			
2. **Elephant Valley** *0.275, 0*			
3. **Triple Spiral Valley** *-0.088,0.654*			
4. **Quad-Spiral Valley** *0.274,0.482*		7. **Double Scepter Valley** *-0.1002,0.8383*	
5. **Scepter Valley** *-1.36,0.005*		8. **Mini Mandelbrot** *-1.75,0*	
6. **Scepter Variant** *-1.108,0.230*		9. **Another Mandelbrot** *-0.1592,-1.0317*	

One simple construction of the Mandelbrot set in tones of blue builds a set of colors with goniometric functions of the number of iterations used to determine the belonging of points to the set: Negative values are not allowed, so the program tests each color components against **0**.

```
for i in range(N):                  #   Building the color set
    blue  = int(sin((2*pi/N)*i)*255)
    green = int(sin((2*pi/N)*i-pi/2)*255)
    red   = int(sin((2*pi/N)*i-pi)*255)
    blue  = max(blue,0)
    ...
```

The colors are stored in a list, called **colorSet**. At the end of the list, a supplemental black color is added, to be assigned to the points that overreach the accepted number of iterations. Then, after calibrating a viewport, each pixel is associated to a complex number (where **x=Re(c)** and **y=Im(c)**), then a sequence is activated, ending either when it diverges or it is considered bounded (after a given number of iterations **N**). The designer used in this course allows the program to train on each single pixel, storing the colors in a **Python** matrix (which is a list of lists) matching the viewport.

In **Python**, complex numbers are quite simple to manage, since the language contains a magic constant for the imaginary unit, named **1j**. The notation can be confusing, but it has a logic: the **1** is necessary to avoid possible conflicts with a variable name, while the letter **j** (instead of the classic **i**) is the standard notation for the imaginary unit in engineering, electronics, and is known to mathematicians. One reason for this choice was given by none less than Guido van Rossum, the inventor of **Python**, who maintains that **i** is more easily mistakable for a number or another variable, while the descender of **j** greatly reduces this risk.

```
y = d.t
while y >= d.b:                     #   Explore each viewport pixel
    row = []
    x = d.l
    while x <= d.r:
        c = x + 1j * y
        z = 0
        j = 0
        while abs(z) <= 2 and j < N:    #   Analyze the sequence
            z = z**2 + c
            j+= 1
        row.append((x,y,colorSet[j]))   #   Store pixel color
        x+= d.px
    pixel.append(row)               #   Store entire row of pixel data
    y-= d.py
```

It is worth mentioning that most solutions found on the Internet use a **for** loop with an horrendous **break** statement activated when the sequence exceeds **2**. Nothing of this sort would be accepted by the masters.

The last step consists in sketching each pixel according to the colors stored. No coordinates are needed, since the colors correspond to the raster background of the viewport:

```
for i in range(len(pixel)):
    for j in range (len(pixel[i])):
        d.screen.create_line(j,i,j+1,i+1,fill=pixel[i][j],width=1)
```

Using the Tkinter **create_line** function instead of a **Pixel** object of the designer cuts the processing time of the viewport in half. Here is how the content of the innermost loop would appear with marioGraphics:

```
p = Pixel(d.wx(j),d.wy(i))
d.draw(p,pixel[i][j])
```

The solution just described, however, is iterative, while the nature of the sequence calls for a recursive one: the problem with such a solution is that the pixel color does not depend on the **z** value of the sequence, but on the number of steps undertaken at the moment the sequence is found to diverge (that is, when **|z|** exceeds **2**). This requires that both values must enter the recursion mechanism: with **Python** this is made possible by using multiple **return** values (something almost unheard of in other languages), so that the recursion stack can be uncoiled without losing track of the recursion step where divergence is detected.

The call to this recursive function should be as simple as possible, with just two arguments needed (the complex number and the maximum number of iterations):

```
color = mandelbrot(c,N)
```

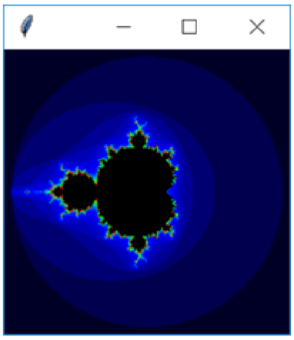
Inside the function a certain complexity is necessary, since there are multiple values involved.

```
def mandelbrot(c,n):
    def benoit(n,esc=False):
        if n < 1:                   #   First iteration: return 0
            return 0,0.,None
        else:
            level,z,esc = benoit(n-1)  #   Activate recursion
            if esc:
                return level,z,True    #   Escape mechanism in progress - exit
            else:                      #   "level" traverse the recursion stack
                z = z ** 2 + c
                esc = abs(z) > 2       #   Check for diverging sequences
                return n,z,esc         #   Yield data to next recursion level
    color,dummy,dummy = benoit(n)      #   Only the color is meaningful here
    return color
```

In the rest of the program, a pair of nested loops (in **y** and **x**) should map al the pixels of the viewport, then used as complex coordinates of the generic number fed to the preceding function, along with the desired number of iterations:

```
c = x + 1j * y
color = mandelbrot(c,N)
```

The resulting colors can be stored in a matrix with a one-to-one correspondence with the viewport, then displayed. At the end the familiar result can be seen:



Of course, the dimensions of the viewport greatly influence the running time of the program. Another trick that has been used was using the **Tkinter** graphic primitives, which cut considerably the running time with respect to the designer. If we consider **i** and **j** (the indexes of the matrix) as coordinates, this translates to:

```
d.screen.create_line(j,i,j+1,i+1,fill=color,width=1)
```

Using the designer in the same point requires screen coordinates and some additional overhead:

```
p = Pixel(d.wx(j),d.wy(i))
d.draw(p,color)
```

The resulting execution times, clocked in some different situations, are summarized in the following table:

Definition	Time in seconds (B/W)	Time in seconds (color)	With Designer
200 x 200	2	3	5
400 x 400	7	12	20
600 x 600	15	22	38

Appendices

(or: Appendixes?)

Appendix A –color names

The **Tkinter**-based designer supports some 480 color words, a number of which correspond to the **HTML** color names. These are summarized in the following table:

ALICEBLUE	DARKORANGE	INDIGO	MEDIUMPURPLE	ROSYBROWN
ANTIQUEWHITE	DARKORCHID	IVORY	MEDIUMSEAGREEN	ROYALBLUE
AQUA	DARKRED	KHAKI	MEDIUMSLATEBLUE	SADDLEBROWN
AQUAMARINE	DARKSALMON	LAVENDER	MEDIUMSPRINGGREEN	SALMON
AZURE	DARKSEAGREEN	LAVENDERBLUSH	MEDIUMTURQUOISE	SANDYBROWN
BEIGE	DARKSLATEBLUE	LAWNGREEN	MEDIUMVIOLETRED	SEAGREEN
BISQUE	DARKSLATEGRAY	LEMONCHIFFON	MIDNIGHTBLUE	SEASHELL
BLACK	DARKSLATEGREY	LIGHTBLUE	MINTCREAM	SIENNA
BLANCHEDALMOND	DARKTURQUOISE	LIGHTCORAL	MISTYROSE	SILVER
BLUE	DARKVIOLET	LIGHTCYAN	MOCCASIN	SKYBLUE
BLUEVIOLET	DEEPPINK	LIGHTGOLDENRODYELLOW	NAVAJOWHITE	SLATEBLUE
BROWN	DEEPSKYBLUE	LIGHTGRAY	NAVY	SLATEGRAY
BURLYWOOD	DIMGRAY	LIGHTGREY	OLDLACE	SLATEGREY
CADETBLUE	DIMGREY	LIGHTGREEN	OLIVE	SNOW
CHARTREUSE	DODGERBLUE	LIGHTPINK	OLIVEDRAB	SPRINGGREEN
CHOCOLATE	FIREBRICK	LIGHTSALMON	ORANGE	STEELBLUE
CORAL	FLORALWHITE	LIGHTSEAGREEN	ORANGERED	TAN
CORNFLOWERBLUE	FORESTGREEN	LIGHTSKYBLUE	ORCHID	TEAL
CORNSILK	FUCHSIA	LIGHTSLATEGRAY	PALEGOLDENROD	THISTLE
CRIMSON	GAINSBORO	LIGHTSLATEGREY	PALEGREEN	TOMATO
CYAN	GHOSTWHITE	LIGHTSTEELBLUE	PALETURQUOISE	TURQUOISE
DARKBLUE	GOLD	LIGHTYELLOW	PALEVIOLETRED	VIOLET
DARKCYAN	GOLDENROD	LIME	PAPAYAWHIP	WHEAT
DARKGOLDENROD	GRAY	LIMEGREEN	PEACHPUFF	WHITE
DARKGRAY	GREY	LINEN	PERU	WHITESMOKE
DARKGREY	GREEN	MAGENTA	PINK	YELLOW
DARKGREEN	GREENYELLOW	MAROON	PLUM	YELLOWGREEN
DARKKHAKI	HONEYDEW	MEDIUMAQUAMARINE	POWDERBLUE	
DARKMAGENTA	HOTPINK	MEDIUMBLUE	PURPLE	
DARKOLIVEGREEN	INDIANRED	MEDIUMORCHID	RED	

There follows a more or less complete list of **Tkinter**-supported colors, along with a sample, found at:

http://www.science.smith.edu/dftwiki/index.php/Color_Charts_for_TKinter

Table 1

AntiqueWhite1 AntiqueWhite2 AntiqueWhite3 AntiqueWhite4 CadetBlue1 CadetBlue2 CadetBlue3 CadetBlue4 DarkGoldenrod1 DarkGoldenrod2 DarkGoldenrod3 DarkGoldenrod4 DarkOliveGreen1 DarkOliveGreen2 DarkOliveGreen3 DarkOliveGreen4 DarkOrange1 DarkOrange2 DarkOrange3 DarkOrange4 DarkOrchid1 DarkOrchid2 DarkOrchid3 DarkOrchid4 DarkSeaGreen1 DarkSeaGreen2 DarkSeaGreen3 DarkSeaGreen4 DarkSlateGray1 DarkSlateGray2 DarkSlateGray3 DarkSlateGray4 DeepPink2 DeepPink3 DeepPink4 DeepSkyBlue2 DeepSkyBlue3 DeepSkyBlue4 DodgerBlue2 DodgerBlue3 DodgerBlue4 HotPink1 HotPink2 HotPink3 HotPink4 IndianRed1 IndianRed2 IndianRed3 IndianRed4 LavenderBlush2 LavenderBlush3 LavenderBlush4 LemonChiffon2 LemonChiffon3 LemonChiffon4 LightBlue1 LightBlue2 LightBlue3 LightBlue4 LightCyan2	LightCyan3 LightCyan4 LightGoldenrod1 LightGoldenrod2 LightGoldenrod3 LightGoldenrod4 LightPink1 LightPink2 LightPink3 LightPink4 LightSalmon2 LightSalmon3 LightSalmon4 LightSkyBlue1 LightSkyBlue2 LightSkyBlue3 LightSkyBlue4 LightSteelBlue1 LightSteelBlue2 LightSteelBlue3 LightSteelBlue4 LightYellow2 LightYellow3 LightYellow4 MediumOrchid1 MediumOrchid2 MediumOrchid3 MediumOrchid4 MediumPurple1 MediumPurple2 MediumPurple3 MediumPurple4 MistyRose2 MistyRose3 MistyRose4 NavajoWhite2 NavajoWhite3 NavajoWhite4 OliveDrab1 OliveDrab2 OliveDrab4 OrangeRed2 OrangeRed3 OrangeRed4 PaleGreen1 PaleGreen2 PaleGreen3 PaleGreen4 PaleTurquoise1 PaleTurquoise2 PaleTurquoise3 PaleTurquoise4 PaleVioletRed1 PaleVioletRed2 PaleVioletRed3 PaleVioletRed4 PeachPuff2 PeachPuff3 PeachPuff4 RosyBrown1	RosyBrown2 RosyBrown3 RosyBrown4 RoyalBlue1 RoyalBlue2 RoyalBlue3 RoyalBlue4 SeaGreen1 SeaGreen2 SeaGreen3 SkyBlue1 SkyBlue2 SkyBlue3 SkyBlue4 SlateBlue1 SlateBlue2 SlateBlue3 SlateBlue4 SlateGray1 SlateGray2 SlateGray3 SlateGray4 SpringGreen2 SpringGreen3 SpringGreen4 SteelBlue1 SteelBlue2 SteelBlue3 SteelBlue4 VioletRed1 VioletRed2 VioletRed3 VioletRed4 alice blue antique white aquamarine aquamarine2 aquamarine4 azure azure2 azure3 azure4 bisque bisque2 bisque3 bisque4 blanched almond blue blue violet blue2 blue4 brown1 brown2 brown3 brown4 burlywood1 burlywood2 burlywood3 burlywood4 cadet blue	chartreuse2 chartreuse3 chartreuse4 chocolate1 chocolate2 chocolate3 coral coral1 coral2 coral3 coral4 cornflower blue cornsilk2 cornsilk3 cornsilk4 cyan cyan2 cyan3 cyan4 dark goldenrod dark green dark khaki dark olive green dark orange dark orchid dark salmon dark sea green dark slate blue dark slate gray dark turquoise dark violet deep pink deep sky blue dim gray dodger blue firebrick1 firebrick2 firebrick3 firebrick4 floral white forest green gainsboro ghost white gold gold2 gold3 gold4 goldenrod goldenrod1 goldenrod2 goldenrod3 goldenrod4 gray gray1 gray10 gray11 gray12 gray13 gray14 gray15

Table 2

gray16	gray71	light sky blue	purple4
gray17	gray72	light slate blue	red
gray18	gray73	light slate gray	red2
gray19	gray74	light steel blue	red3
gray2	gray75	light yellow	red4
gray20	gray76	lime green	rosy brown
gray21	gray77	linen	royal blue
gray22	gray78	magenta2	saddle brown
gray23	gray79gray8	magenta3	salmon
gray24	gray80	magenta4	salmon1
gray25	gray81	maroon	salmon2
gray26	gray82	maroon1	salmon3
gray27	gray83	maroon2	salmon4
gray28	gray84	maroon3	sandy brown
gray29	gray85	maroon4	sea green
gray3	gray86	medium aquamarine	seashell2
gray30	gray87	medium blue	seashell3
gray31	gray88	medium orchid	seashell4
gray32	gray89	medium purple	sienna1
gray33	gray9	medium sea green	sienna2
gray34	gray90	medium slate blue	sienna3
gray35	gray91	medium spring green	sienna4
gray36	gray92	medium turquoise	sky blue
gray37	gray93	medium violet red	slate blue
gray38	gray94	midnight blue	slate gray
gray39	gray95	mint cream	snow
gray4	gray97	misty rose	snow2
gray40	gray98	navajo white	snow3
gray42	gray99	navy	snow4
gray43	green yellow	old lace	spring green
gray44	green2	olive drab	steel blue
gray45	green3	orange	tan1
gray46	green4	orange red	tan2
gray47	honeydew2	orange2	tan4
gray48	honeydew3	orange3	thistle
gray49	honeydew4	orange4	thistle1
gray5	hot pink	orchid1	thistle2
gray50	indian red	orchid2	thistle3
gray51	ivory2	orchid3	thistle4
gray52	ivory3	orchid4	tomato
gray53	ivory4	pale goldenrod	tomato2
gray54	khaki	pale green	tomato3
gray55	khaki1	pale turquoise	tomato4
gray56	khaki2	pale violet red	turquoise
gray57	khaki3	papaya whip	turquoise1
gray58	khaki4	peach puff	turquoise2
gray59	lavender	pink	turquoise3
gray6	lavender blush	pink1	turquoise4
gray60	lawn green	pink2	violet red
gray61	lemon chiffon	pink3	wheat1
gray62	light blue	pink4	wheat2
gray63	light coral	plum1	wheat3
gray64	light cyan	plum2	wheat4
gray65	light goldenrod	plum3	white smoke
gray66	light goldenrod	plum4	yellow
gray67	yellow	powder blue	yellow green
gray68	light grey	purple	yellow2
gray69	light pink	purple1	yellow3
gray7	light salmon	purple2	yellow4
gray70	light sea green	purple3	

Appendix B – summary of geometric tools

SUMMARY OF METHODS OF GEOMETRIC OBJECTS

Examples with points **p** and **q**, segment **s**, lines **r**, **r1**,and, **r2**, and circle **c**.

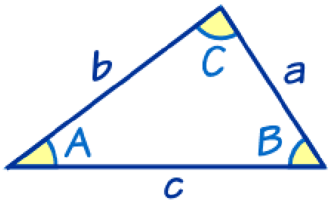
x is a generic name when the method can be used with any shape.

```
s = p.parallel(r)
t = p.perpendicular(r)
t = p.perpendicular(s)    # also with segment s
z = p.projection(r)       # r is a Line: yields a Point
z = p.projection(s)       # s is a Segment: yields a Point
s = p.projection(c)       # c is a Circle: yields a Segment
s2 = p.projection(c)      # point and circle: yields a Segment
z = p.projection(q,d)     # q is another point, d=distance: yields a Point
z1 = r1.section(r2)       # Point joining two lines
z2 = p.section(q)         # Line joining two points
s = r.section(c)          # Line and Circle: yields a Segment

s.rotate( angle)          # rotate a segment on point a
s.stretch( distance)      # move point b to match the new length

p.on(x);                  # p:Point; x=any shape (True or False)
p.internal(c)             # Point and Circle (True=inside, False=outside)

r1.parallet1(r2)          # conditions on lines (True or False)
r.vertical()
r.horizontal()

p.distance(x)             # p=Point; x=any shape
```

TRIANGLES

Sum of angles

```
A + B + C = pi
```



Laws of sines

```
a / sin(A) = b / sin(B) = c / sin(C)
```

Laws of cosines

cos(A) = (b^2 + c^2 – a^2) / (2bc)

SSS (given 3 sides: a, b, and c)

From the law of cosines:

a^2 = (b^2 + c^2 – 2*b*c*cos(A))
A = acos((b^2 + c^2 – a^2) / (2*b*c))

SAS (given 2 sides and the angle in between: a, b, and C)

Use law of cosines:

c = pow(a^2 + b^2 – 2*a*b*cos(C), 0.5)
A = acos((b^2 + c^2 – a^2) / 2bc)
B = acos((a^2 + c^2 – b^2) / 2ac)
C = acos((a^2 + b^2 – c^2) / 2ab)

Square triangles (C = right angle, c = hypotenuse)

```
a = c * cos(B) = c * sin(A)
b = c * cos(A) = c * sin(B)
A = asin(a/c) = acos(b/c)
B = asin(b/c) = acos(a/c)
```

Appendix C – a solution for the linear system problem

In matrix form, a linear system is represented like this:

$$\begin{bmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \dots & & & \\ a_{n1} & a_{n2} & \dots & a_{nn} \end{bmatrix} * \begin{bmatrix} b_1 \\ b_2 \\ \dots \\ b_n \end{bmatrix} = \begin{bmatrix} x_1 \\ x_2 \\ \dots \\ x_n \end{bmatrix}$$

Normally, though, all the known coefficients `b` are attached to matrix `a`, which turns into a `n*(n+1)` matrix, with the known coefficients in the last column.

The following is a solution which uses the Gauss-Banachiewicz method. The indexes are all shifted, as Python lists[77] are zero-based. In matrix notation, the system becomes:

$$\begin{bmatrix} a_{0,0} & a_{0,1} & \dots & a_{0,n-1} \\ a_{1,0} & a_{1,1} & \dots & a_{1,n-1} \\ a_{2,0} & a_{2,1} & \dots & a_{2,n-1} \\ \dots & \dots & \dots & \dots \\ a_{n-1,0} & a_{n-1,1} & \dots & a_{n-1,n-1} \end{bmatrix} * \begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ \dots \\ b_{n-1} \end{bmatrix} = \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ \dots \\ x_{n-1} \end{bmatrix}$$

```
def system(a):
    m = len(a)
    n = len(a[0])
    _a = [[a[i][j] for j in range(n)] for i in range(m)]

    x = [None for y in range(m)]            #   the array for the solutions
    for h in range(m):
        g = h                               #   1: finding the pivot
        for i in range(h+1,m):
            if abs( _a[i][h]) > abs( _a[g][h]):
                g = i

        if g != h:                          #   2: place pivot in place
            for j in range(n):
                (_a[g][j],_a[h][j])=(_a[h][j],_a[g][j])

        if _a[h][h] != 0:                   #   3: linear combination
            for i in range(h+1,m):          #   (using rectangle rule)
                z = -_a[i][h] / _a[h][h];
                for j in range(n):
                    _a[i][j]+= z * _a[h][j]

    for i in range(m-1,-1,-1):              #   4: solution
        for j in range(i+1,m):              #   (using back-substitution)
            _a[i][n-1]-= x[j] * _a[i][j]
            _a[i][j] = 0;
        x[i] = _a[i][n-1] / _a[i][i]        #   value of each unknown

    return x
```

[77] *List* is the Python term for *array*.

Appendix D – a bit of recursion

Example 1

Writing a base sequence recursively is a walk in the park:

```
def display1(n):            #   straight
    if n>0:
        display1(n-1)
        write(n)

def display2(n):            #   reversed
    if n>0:
        write(n)
        display2(n-1)
```

> What is remarkable here is how the output statement is deferred until the end of the recursion chain.

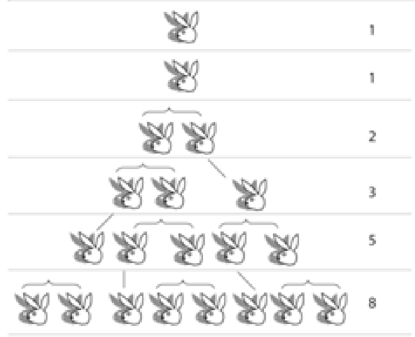
Example 2

One of the “most recursive” mathematical functions is the definition of the Fibonacci[78] sequence, where each term is computed as the sum of the preceding two (the first two terms being **1**). It is the solution to the problem of how many offspring can descend from a couple of rabbits (this explains the first two terms). The function is a spitting image of the definition given in math courses:

```
#--- mathematical definition
```

$$f_0 = 1$$
$$f_1 = 1$$
$$f_n = f_{n-1} + f_{n-2}$$

```
#--- Python function

def fibonacci(n):
    if n<=1:
        return 1
    else:
        return fibonacci(n-1) + fibonacci(n-2)
```



Fibonacci's rabbits

For a test:

```
for i in range(6):
    write(fibonacci(i))

>>> 1 1 2 3 5 8
```

Example 3

This is how to find recursively the largest element in a collection:

```
def findMax(L):
    if len(L) == 1:
        x = L[0]
    else:
        x = max( L[0], findMax(L[1:]))
    return x
```

For a test:

```
print(findMax('mackerel'))

>>> r
```

Of course, given that Python already has a function called `max()` which does exactly this, it is more of an example than something really useful.

---

78 Leonardo of Pisa (1175?-1235?), known as Fibonacci (son of Bonaccio, his father being a Guglielmo Bonacci), one of the key figures in the history of mathematics, was instrumental in the building of algebra and modern mathematics.